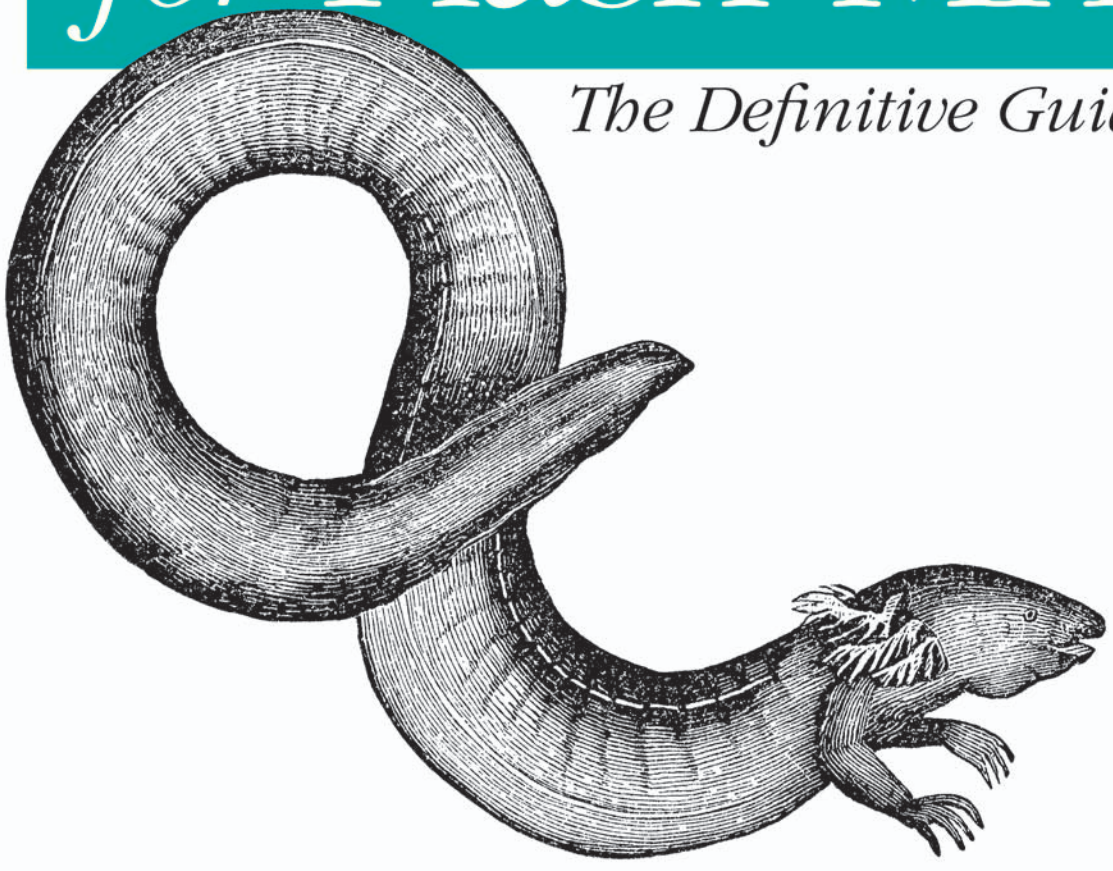
Mastering Flash MX Programming

2nd Edition

Foreword by Gary Grossman
Creator of ActionScript

ActionScript *for* Flash MX

The Definitive Guide



O'REILLY®

Colin Moock

O'REILLY®

ActionScript for Flash MX: The Definitive Guide

Macromedia Flash MX is the de facto standard for delivering web-based multimedia to over 500 million users worldwide. *ActionScript for Flash MX: The Definitive Guide* is dedicated entirely to ActionScript, Flash MX's object-oriented programming language. This book is for web developers using Flash for true application development, not just eye candy. The book targets Flash and ActionScript developers of all levels, plus JavaScript programmers migrating their skills to ActionScript.

Updated to cover Flash MX, this second edition is the one book no serious Flash developer should be without. It covers new features in Flash MX ActionScript, such as the Drawing API, loading of external MP3 and JPEG files, improved sound control, text formatting, component development using movie clip subclasses, local data storage, accessibility features, and much more. *ActionScript for Flash MX: The Definitive Guide* is the most complete, up-to-date reference available, with over 400 new pages added since the best-selling first edition. The book's Language Reference has nearly doubled in size, with more than 250 new classes, objects, methods, and properties. You'll find exhaustive coverage of dozens of undocumented, under-documented, and misdocumented features.

Author Colin Moock, one of the most universally respected developers in the Flash community, has added hundreds of new code examples to show new Flash MX techniques in the real world: how to draw shapes at runtime, save data to disk, convert arrays to onscreen tables, create reusable components, and preload variables, XML, and sounds. His meticulously revised text conforms to Flash MX best-coding practices, and he gives special attention to objected-oriented programming and the new Flash MX event model.

"ActionScript for Flash MX: The Definitive Guide *is unquestionably the essential book for ActionScript programming in Flash MX.*"

—Gary Grossman, Creator of ActionScript, Macromedia

"Colin has an incredible mastery of ActionScript and is able to convey his knowledge in a clear, concise, and often witty manner. I can't praise this book enough."

—Branden J. Hall, co-author of *Object-Oriented Programming with ActionScript*

"Time and time again, I recommend only one book: Colin Moock's ActionScript for Flash MX: The Definitive Guide*."*

—Joshua Davis, Artist and Technologist (*http://www.praystation.com*), author of *Flash to the Core*

www.oreilly.com

US $54.95 **CAN $85.95**
ISBN-10: 0-596-00396-X
ISBN-13: 978-0-596-00396-8

Praise for ActionScript for Flash MX: The Definitive Guide

"Like its predecessor, *ActionScript for Flash MX: The Definitive Guide,* Second Edition (*ASDG2*) is *the* must-have book for ActionScript developers. Moock has delivered a comprehensive revision that not only details the myriad new features in Flash MX but explains the underlying shifts in methodology (especially with components and events) in a clear and accessible manner."

—Robert Penner, Author of *Robert Penner's Programming Macromedia Flash MX*

"I love this book. It's an approachable, yet incredibly in-depth exploration of ActionScript. If you're working with Flash MX—designer or programmer—this is a must have."

—Hillman Curtis, Founder of hillmancurtis, Inc. (*http://www.hillmancurtis.com*), Author of *Flash Web Design* and *MTIV: Process, Inspiration and Practice for the New Media Designer*

"You can't talk about this book without resorting to superlatives. Moock's *ASDG2* is the most essential book on Flash programming you can buy. An exhaustive and indispensable reference, it pays equal attention to standard cases, best practices, and hidden gotchas. There is no ActionScripter who can't learn more about the language through reading it."

—Nigel Pegg, Component Engineer, Macromedia (Developer of the Flash UI Components)

"As a developer, instructor, and fellow author, I can't praise *ASDG2* enough. This book is simply indispensable for anyone working with Macromedia Flash. Colin has an incredible mastery of ActionScript and is able to convey his knowledge in a clear, concise, and often witty manner."

—Branden J. Hall, Flashcoders List Founder/Admin, Author of *Object-Oriented Programming with ActionScript*

"Colin has once again created a must-have reference for Flash developers. The in-depth coverage of Flash MX ActionScript makes this the most comprehensive resource and reference available. *ASDG2* will once again be my reference of choice."

—Dave Yang, CEO and Developer, Quantumwave Interactive Inc.

"When the first edition of this book came out, it included information that simply did not exist anywhere else. This is twice as true for the second edition, which now deals with the ever more complex Flash MX. Once again, Moock and O'Reilly have produced the best-written, most organized, and meticulously proofed resource for Flash MX developers of all levels. Whether you are a designer, programmer, or artist, this book is invaluable to your development and expression with Flash MX."

—Amit Pitaru, Artist and Technologist (*http://www.pitaru.com*, *http://www.insertsilence.com*), Pratt Institute & NYU (*http://itp.nyu.edu*) Instructor, contributor to *New Masters of Flash: The 2002 Annual*

"This book is the 'End All' ActionScript resource by the man who quite frankly started it all."

— Todd Purgason, Creative Director and Co-founder of Juxt Interactive (*http://www.juxtinteractive.com*), Author of *Flash Deconstruction*

"*ASDG2* is unquestionably the essential book for ActionScript programming in Flash MX. It has proven invaluable even for the engineers on the Macromedia Flash team, who see it as complementary to our own product documentation. Moock's meticulous attention to detail, evident throughout this fine volume, combined with his easygoing instructional style, ensure the book will be appreciated by newcomers and experts alike."

— Gary Grossman, Creator of ActionScript, Macromedia

"*ASDG2* is a brilliantly concise and direct resource. It is broad and thorough, leaving practically no stone in Flash unturned. It is the only Flash book that I own, and there are many times that I would have been left tearing out my hair without it. Something about the way that Colin structures information and ideas makes even the most gruesomely technical regions of Flash seem natural to me."

— James Patterson, Artist (*http://www.presstube.com*), contributor to *New Masters of Flash*

"*ASDG2* is the only place to go to learn how to take advantage of everything Flash MX can actually do. Even Macromedia doesn't go into the detail that Colin Moock does. Whether you are taking a week to learn ActionScript or have an hour to work around a problem, *ASDG2* is the only reference that will get you through it. Best of all, Moock is a wonderful writer, and he's excited! Never has a book on programming been so enjoyable to read."

— Josh Ulm, Founder of ioResearch Studios (*http://www.ioresearch.com*) and The Remedi Project (*http://www.theremediproject.com*)

"Moock's book is the one that never leaves the desktop. Whether you need answers in the middle of a tight project deadline or are looking to deepen your coding knowledge, it's the invaluable guide to the ins and outs of ActionScript."

— Glenn Thomas, Director and Founder, Smashing Ideas, Inc. (*http://www.smashingideas.com*), Author, *Flash Studio Secrets*, Co-author of *Flash Enabled*

"*ASDG2* is quite simply the most precise, well constructed book on Flash programming in print. Colin Moock obviously has a passion for getting it right."

— Jared Tarbell, Computational Artisan, (*http://www.levitated.net*), contributor to *Flash Math Creativity* and *Fresh Flash*

"While there's a vast array of Flash books on the market (including my own!), time and time again, I recommend only one book: Colin Moock's *ASDG2*. Whether on its own or as a companion to any other Flash book you might buy, Colin's book provides a definitive holy grail of new concepts for Flash professionals around the world, and helps us keep our syntax correct!"

— Joshua Davis, Artist and Technologist (*http://www.praystation.com*), Author of *Flash to the Core*

ActionScript for Flash MX

The Definitive Guide

SECOND EDITION

ActionScript for Flash MX

The Definitive Guide

Colin Moock

O'REILLY®

Beijing • Cambridge • Farnham • Köln • Paris • Sebastopol • Taipei • Tokyo

ActionScript for Flash MX: The Definitive Guide, Second Edition
by Colin Moock

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly Media, Inc. books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*safari.oreilly.com*). For more information, contact our corporate/institutional sales department: (800) 998-9938 or *corporate@oreilly.com*.

Editor:	Bruce Epstein
Production Editor:	Brian Sawyer
Cover Designer:	Ellie Volckhausen
Interior Designer:	David Futato

Printing History:

May 2001:	First Edition. Originally published under the title *ActionScript: The Definitive Guide*.
December 2002:	Second Edition.

Nutshell Handbook, the Nutshell Handbook logo, and the O'Reilly logo are registered trademarks of O'Reilly Media, Inc. *ActionScript for Flash MX, Second Edition*, the image of a siren, and related trade dress are trademarks of O'Reilly Media, Inc. Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and O'Reilly Media, Inc. was aware of a trademark claim, the designations have been printed in caps or initial caps.

While every precaution has been taken in the preparation of this book, the publisher and author assume no responsibility for errors or omissions, or for damages resulting from the use of the information contained herein.

ISBN: 0-596-00396-X
[M] [5/06]

Table of Contents

Part II. Language Reference

Part III. Appendixes

Foreword

A scant eighteen months have passed since I penned the Foreword for the first edition of *ActionScript: The Definitive Guide*. Since that time, the first edition has established itself as the essential guide to ActionScript programming. It's become so indispensable to so many developers that it seems as if it has existed for a much longer time.

Flash MX, which shipped in March 2002, was the most ambitious release of Flash to date. The team of talented individuals that contributed to its creation was larger than ever, and we delivered over 100 major new features. ActionScript was a key focus, which required a change in the way it was developed. Prior to Flash MX, ActionScript was developed by a handful of individuals, including myself. In MX, our ambitious ActionScript agenda required many engineers. With the additional resources, we were able to deliver a vastly improved script editor and debugger, optimize performance, and add a plethora of new APIs providing new capabilities for ActionScript programmers.

There is a great deal of excitement about Flash at Macromedia today. While the public may think of Flash as simply an animation tool, the Flash developer community is beginning to recognize that Flash is something broader. With Flash MX, web developers now have the means to deliver rich, interactive user experiences over the Web—not only the traditional uses of Flash, such as cartoons and motion graphics, but also sophisticated web applications.

Flash always has been, and seems destined to remain, the best way to give your web site some pizzazz, but serious web application developers are straining against the limitations of HTML. They are searching for a new platform that offers more attractive, engaging, and usable experiences to their users—a *rich client*—and they are finding Flash to be an ideal delivery vehicle. Flash's cross-platform consistency and ubiquitous distribution base offer a runtime technology upon which developers can build a new breed of web applications that are more interesting and nimble than those that existed previously. I'd wager that you'll be seeing a broad spectrum of new uses for Flash, from multiplayer games to e-commerce to data visualization. And

Macromedia is committed to ensuring that Flash keeps up with the new demands placed on it by application developers. ActionScript plays an important role in this new vision for Flash MX. Because the usefulness of the Flash platform depends on the power of its scripting language, we set out to make ActionScript powerful enough to satisfy even the most ambitious web developer.

This initiative to make Flash a true application platform posed special challenges for developing Flash MX. Flash is, in a sense, a product being pulled in many directions at once, as it addresses the needs of many different customers, from character animation to motion graphics to the growing field of rich application development. Scripting enhancements were seen as critical, but we realized that it was equally important to enhance Flash's abilities for creative expression, because visual artistry is the heart and soul of Flash.

To ensure that we fulfilled the varied needs of our customers, we divided the Flash engineering team into three groups, each with its own mandate:

Approachable
: Provide an excellent initial experience for new users

Creative
: Enhance Flash's abilities of creative expression

Power
: Beef up ActionScript into a powerful tool for developing complex applications

I was delighted to lead the *Power* team, which went about enhancing ActionScript to support the notion of "Flash as a platform." We revised and enhanced Flash's object and event models; we refined Flash 5 Smart Clips into a more robust component architecture; and we rewrote frequently used ActionScript objects to optimize performance. In addition, we added power tools for developers, such as Code Hints and the revamped Debugger.

We weren't the only ones working on ActionScript, however. The union of Macromedia and Allaire in 2001 brought the company formidable server expertise. The folks at the new Macromedia office in Newton, Massachusetts built Macromedia Flash Remoting MX (Flash Remoting), a new server-side technology permitting direct and easy-to-use communication with the back end. The all-stars on the Macromedia Flash Communication Server MX (Comm Server) team pushed the envelope on what can be done with ActionScript, introducing new ActionScript APIs (including ServerSide ActionScript) that enable truly trailblazing functionality: live two-way communications and collaboration over the Internet!

Another entire team was dedicated to the task of building components. The Components Team—of which two members served as technical editors for this book—built UI components that enable the quick construction of HTML-like forms, and additional controls that go beyond what is possible with HTML, such as a full-blown tree

control, calendar control, and a data grid. Combined with Flash Remoting, the components are a formidable force for building data-driven applications.

The components in Flash MX offer a potent taste of the future: high-level abstractions that can quickly be assembled into interactive content and applications. At Macromedia, we will seek to make the construction and usage of components easier and even more powerful in future releases of Flash. The components offered with Comm Server are a great example of that power. Even without components, using Comm Server, it is relatively easy to build a videoconferencing application in only a few lines of ActionScript. Comm Server components make it even easier; by simply dragging a few components, novices can effectively script without using ActionScript. This is the direction we're interested in, because it helps novice users become productive immediately. Rest assured that as ActionScript and Flash become more approachable, greater possibilities will open up for advanced developers. By taking care of the mundane plumbing and commonly used UI components, we enable expert users and programmers to be even more productive. Flash MX's enhanced object model and component architecture allows skilled developers to extend existing components or develop their own custom libraries. So, whereas this book doesn't cover the existing components in detail, it offers advanced and aspiring developers the tools to create their own. It is always exciting to see the new directions developers take ActionScript once they have the tools and an understanding of how to use them.

Therefore, this second edition is unquestionably the essential book for ActionScript programming in Flash MX. It has proven invaluable even for the engineers on the Macromedia Flash team, who see it as complementary to our own product documentation. This book is the product of Colin Moock's boundless talent and energy, which have driven him to delve deeply into ActionScript, probing its inner secrets for your benefit. His meticulous attention to detail, evident throughout this fine volume, combined with his easygoing instructional style, ensure the book will be appreciated by newcomers and experts alike. Enjoy the book, and enjoy ActionScript in Flash MX!

—Gary Grossman
Creator of ActionScript
Senior Engineering Manger
Macromedia Flash Team
October 2002

Preface

Welcome to *ActionScript for Flash MX: The Definitive Guide,* Second Edition! This edition sports massive changes from the first edition, with hundreds of pages of new material and exhaustive rewrites that bring old material up to date with best practices for Flash MX. I hope you're as excited to read it as I was to write it!

Like the first edition, this book teaches ActionScript from the ground up, covering both basic concepts and advanced usage, but with a special focus on Macromedia Flash MX techniques. In Part I, we'll explore ActionScript fundamentals—from variables and movie clip control to advanced topics such as objects, classes, and server communication. In Part II, the *Language Reference*, we'll cover every object, class, property, method, and event handler in the core ActionScript language. You'll use the *Language Reference* regularly to learn new things and remind yourself of the things you always forget, so keep this book on your desk, not on your shelf!

Though ActionScript's complexity has increased in Flash MX, you do not have to be a programmer to read this book. I have continued to be mindful of the beginner throughout this edition. The text moves pretty quickly, but a prior knowledge of programming is not required to read it. All you need is experience with the non-ActionScript aspects of Flash and an eagerness to learn. Of course, if you are already a programmer, so much the better; you'll be applying your code-junkie skills to ActionScript in no time. To make the transition to Flash easier for experienced programmers, I've made a special effort to draw helpful analogies to languages such as JavaScript, Java, and C.

Above all, this book truly is a Definitive Guide to ActionScript in Flash MX. It's the product of nearly four years of research, thousands of emails to Macromedia employees, and feedback from users of all levels. I hope that it is self-evident that I've suffused the book with both my intense passion for the subject and the painfully won, real-world experience from which you can benefit immediately. It covers ActionScript with exhaustive authority and—thanks to a technical review by Gary Grossman, the creator of ActionScript—with unparalleled accuracy.

Second Edition Quick Start

If you're a returning first-edition reader dying to sink your teeth into this edition, here are the highlights I recommend you start with. But don't end your exploration with this list. Read on to learn about many more important updates to this edition.

The following chapters in Part I, *ActionScript Fundamentals*, have been heavily rewritten and enhanced. They cover some of the most exciting additions, such as components, and meaningful changes to the way ActionScript handles events and deals with objects.

- Chapter 9, *Functions*
- Chapter 10, *Events and Event Handling*
- Chapter 12, *Objects and Classes*
- Chapter 14, *Movie Clip Subclasses and Components*

See also the revised and new appendixes, especially:

- Appendix C, *Backward Compatibility and Player Build Updates*
- Appendix E, *HTML Support in Text Fields*
- Appendix F, *Support for GET and POST*
- Appendix G, *Flash UI Component Summary*
- Appendix H, *Embedding a Flash Movie in a Web Page*

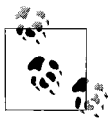
The following entries in Part II, the *Language Reference*, are either all-new or have been heavily revised since the first edition. For example, you'll want to read up on the new *SharedObject* object and check out the Drawing API methods added to the *MovieClip* class.

- *Accessibility* object
- *Button* class
- *Capabilities* object
- *Function* class
- `_global` object
- `#initclip` and `#endinitclip` pragmas
- *LoadVars* class
- *LocalConnection* class
- *MovieClip* class (new events and the Drawing API)
- *Object* class
- *setInterval()* and *clearInterval()* global functions
- *SharedObject* object
- *Sound* class

- *Stage* object
- *System* object
- *TextField* class
- *TextFormat* class
- Listener Events for *Key*, *Mouse*, *TextField*, and *Stage* (see Table P-1)

What's New in Flash MX ActionScript

ActionScript evolved tremendously from Flash 5 to Flash MX (as the authoring tool is known) and the corresponding Flash Player 6, and this book has evolved along with it. See Table P-2 in this Preface for details on the Flash version naming conventions.



To preview many of the new features in action, visit:
http://www.moock.org/webdesign/lectures/newInMX

Table P-1 provides a high-level overview of the major additions to ActionScript and tells you where to find more information about each new topic in this book. Unless otherwise stated, cross-references are to Part II, the *Language Reference*.

Table P-1. New features in Flash MX ActionScript

Feature	For details, see...
Drawing API: draw strokes, shapes, and fills at runtime using new *MovieClip* methods	*MovieClip.beginFill( )*, *MovieClip.beginGradientFill( )*, *MovieClip.clear( )*, *MovieClip.curveTo( )*, *MovieClip.endFill( )*, *MovieClip.lineStyle( )*, *MovieClip.lineTo( )*, *MovieClip.moveTo( )*; "Drawing in a Movie Clip at Runtime" in Chapter 13
Load JPEG-format images at runtime	*MovieClip.loadMovie( )*, *loadMovie( )*
Load MP3-format sounds at runtime	*Sound.loadSound( )*
Check the length of a sound and the amount of time it has been playing	*Sound*.position, *Sound*.duration
Detect when a sound finishes playing	*Sound.onSoundComplete( )*
Create, manipulate, and format text fields at runtime	The *TextField* class, the *TextFormat* class, *MovieClip.createTextField( )*
Mask or unmask a movie clip at runtime	*MovieClip.setMask( )*
Create movie clips from scratch at runtime	*MovieClip.createEmptyMovieClip( )*
Determine a movie clip's depth at runtime	*MovieClip.getDepth( )*
Execute a function or method periodically	*setInterval( )*, *clearInterval( )*
Manipulate XML, string, and array data faster due to Flash Player performance improvements	The *XML* class, the *String* class, the *Array* class
Store data locally (much like JavaScript cookies)	The *SharedObject* object

Table P-1. New features in Flash MX ActionScript (continued)

Feature	For details, see...
Create packaged code modules with *MovieClip* subclasses and components	`#initclip`, `#endinitclip`, *Object.registerClass( )*, *attachMovie( )*; Chapter 14
Communicate between two Flash Players on the same computer	The *LocalConnection* class
Declare global variables	`_global`; "Movie Clip Variables and Global Variables" in Chapter 2
Use international characters in the Unicode character set	"The String Type" in Chapter 4, Appendix C
Define event handlers on movie clips using callback functions	Chapter 10
Use event listeners to respond to events from any object	Chapter 10 and *Key.addListener( )*, *Mouse.addListener( )*, *Stage.addListener( )*, *Selection.addListener( )*, *TextField.addListener( )*
Add button behavior to a movie clip	"Using Movie Clips as Buttons" in Chapter 13
Control button objects at runtime	The *Button* class
Make content accessible to screen readers for the visually impaired	The *Accessibility* object
Check the movie width and height at runtime, and reposition movie elements when the movie is resized	`Stage.height`, `Stage.width`, *Stage.onResize( )*
Use lexical and nested function scope, or execute a function as a method of an arbitrary object	*Function.call( )*, *Function.apply( )*; "The Scope Chain" in Chapter 2; "Function Scope" in Chapter 9
Access Player and system information such as screen resolution, operating system, and current language	The *Capabilities* object
Capture keyboard and mouse input events with a centralized input API	The *Key* object, the *Mouse* object
Load variables using an intuitive variable loading class rather than the *loadVariables( )* function	The *LoadVars* class
Monitor the download progress of XML or loading variables	*XML.getBytesLoaded( )*, *LoadVars.getBytesLoaded( )*
Control the tab order for buttons, text fields, and movie clips	*`TextField`*`.tabIndex`, *`Button`*`.tabIndex`, *`MovieClip`*`.tabIndex`
Turn off the hand cursor for buttons	*`Button`*`.useHandCursor`, *`MovieClip`*`.useHandCursor`
Add getter/setter properties to an object, and receive notification when a property changes	*Object.addProperty( )*, *Object.watch( )*

What's New in the Second Edition

The second edition of *ActionScript for Flash MX: The Definitive Guide* is not merely a "tack-on" update to the first edition (which was titled *ActionScript: The Definitive Guide*). The entire text has been revised and restructured to highlight the latest Flash MX ActionScript features. Nearly every paragraph has been updated, and 400 pages have been added to cover ActionScript's new capabilities. Legacy descriptions of Flash 4 ActionScript syntax have been moved from the body of the book to

Appendix C or online technotes. We made this choice to keep the book streamlined, although it is still considerably beefier than the first edition. By the time you read this, Flash Player 6 will be nearly ubiquitous, so it doesn't make sense to cover Flash 4 in detail anymore. We cover enough of it to help you understand and upgrade any legacy code you may own or encounter. We've also paid close attention to changes between Flash 5 and Flash 6 to help you understand the new paradigms and upgrade legacy code. The legacy code examples from the first edition will all remain available at *http://www.moock.org/asdg/codedepot*.

Updated Code Examples

All code examples from the first edition have been rewritten to use Flash MX syntax and best practices. For example:

- The quiz samples now use callback functions—rather than Flash 5–style *on()* handlers—for button event handlers.
- Text fields that were formerly drawn in the authoring tool are now generated programmatically with *createTextField()*.
- Classes are defined on `_global` (the new property that holds global variables)
- The object-oriented *LoadVars* class is used instead of the older *loadVariables()* global function.

Likewise, dozens of new Flash MX-specific examples have been added. Here are just a few of the interesting ones:

- A completely code-based, object-oriented quiz, downloadable from the online Code Depot (described later in this Preface)
- A configurable text ticker (see *`TextField`*`.hscroll`)
- An array-to-table converter (see *`TextFormat`*`.tabStops`)
- A sound preloader (see *Sound.getBytesLoaded()*)

Hundreds of Tweaks

Subtle details have been added throughout this book to augment the first edition's content. Here are just a few of the hundreds of tweaks made:

- *`MovieClip`*`._x` discusses *twips* (the minimum distance a clip can be moved).
- *`MovieClip`*`._visible` warns that button events don't fire when `_visible` is `false`.
- *XML.parseXML()* covers CDATA and predefined XML entities (&, <, >, ", and ') at length.
- *MovieClip.getBytesLoaded()* features a list of possible return values based on the asynchronous execution of *loadMovie()*.

- Chapter 2 discusses qualified and unqualified variable references and *Hungarian notation*.
- Chapter 4 explicitly contrasts `null` with *delete* and `undefined`.

Of course, there are plenty of not-so-subtle changes too. We'll look at them next.

Major Revisions Since the First Edition

The following list describes the major content and structural changes in this second edition. Note that some of these chapters were in Part II, *Applied ActionScript*, in the first edition. Other material from the first edition's Part II was redistributed elsewhere in this second edition, and some content was moved to online technotes. Despite the organizational change, rest assured that this second edition includes dozens of applied examples sprinkled liberally throughout the entire book. The *Language Reference*, formerly Part III in the first edition, is now Part II.

Chapter 1, *A Gentle Introduction for Nonprogrammers*

- Added an introduction to object-oriented programming
- Revised the quiz tutorial for Flash MX
- Revised the event handler section for Flash MX

Chapter 2, *Variables*

- Added recommended suffixes for variable names
- Added global variable coverage
- Added a section on loading external variables
- Added an explicit discussion of the *scope chain*

Chapter 3, *Data and Datatypes*

- Added the section "Copying, Comparing, and Passing Data" (formerly in Chapter 15)

Chapter 4, *Primitive Datatypes*

- Added coverage of Unicode

Chapter 5, *Operators*

- Added coverage of the strict equality and *instanceof* operators

Chapter 6, *Statements*

- Added *switch* statement coverage
- Revised the description of *with* to include the scope chain
- Removed the legacy *call* statement (now covered in the *Language Reference* only)

Chapter 8, *Loop Statements*

- Added a section on using *setInterval()* to execute code repeatedly
- Revised "Timeline and Clip Event Loops" to use Flash MX features (*MovieClip.createEmptyMovieClip()* and the *MovieClip.onEnterFrame()* handler)

Chapter 9, *Functions*

- Added a section on the differences between function literals and the *function* statement
- Added coverage of nested functions
- Revised "Function Scope" to cover *lexical scope* in more detail
- Revised the quiz tutorial for Flash MX

Chapter 10, *Events and Event Handling*

- Added complete coverage of event handler properties
- Added coverage of event listeners, new in Flash MX
- Added an in-depth discussion of scope, including Table 10-1, which compares old scope rules to new scope rules
- Added a description of the `this` keyword within various handlers, including a summary in Table 10-2
- Moved all specific button and movie clip event descriptions to the *Language Reference* (see also Table 10-3)

Chapter 11, *Arrays*

- Added coverage of the *Array.sortOn()* method
- Revised the quiz tutorial for Flash MX

Chapter 12, *Objects and Classes*

- Revised the chapter entirely to focus more squarely on the process of making a class with methods and properties
- Added coverage of Flash MX's *super* keyword, used to invoke a superclass constructor and its methods
- Added a formal discussion of the *prototype chain*
- Added a formal discussion of issues with standard *superclass assignment*
- Added a section on static methods and properties
- Added a description of rendering an object to screen
- Added an object-oriented programming (OOP) application template
- Added an "OOP Quick Reference" section
- Added a brief discussion of UML and design patterns

Chapter 13, *Movie Clips*

- Added information on creating a blank movie clip from scratch using *MovieClip.createEmptyMovieClip( )*
- Added a section on drawing in a movie clip at runtime using the new Drawing API
- Added a section on implementing button behavior for a movie clip
- Added a section on handling input focus for movie clips
- Revised (fixed) the first edition's partially erroneous description of *MovieClip.duplicateMovieClip( )* depths
- Moved the list of *MovieClip* methods and properties to the *Language Reference*
- Moved the legacy *Tell Target* discussion to Appendix C
- Updated the clock example to use Flash MX best practices
- Removed the quiz example, which is superseded by the new downloadable OOP quiz (the legacy version is still available online)

Chapter 14, *Movie Clip Subclasses and Components* (all new)

- Covers how to make movie clip subclasses (specialized types of movie clip symbols associated with a class)
- Covers how to create a basic component, of which the Flash UI Components are a complex example

Chapter 15, *Lexical Structure* (previously Chapter 14)

- Revised the list of reserved words
- Removed and redistributed old Chapter 15, content as follows:
  - Moved "Copying, Comparing, and Passing Data" to Chapter 3
  - Moved "Bitwise Programming" to online technote at *http://www.moock.org/asdg/technotes*
  - Removed "Advanced Function Scope Issues" (the issue discussed was fixed in Flash MX)
  - Moved "The MovieClip Datatype" to online technote at *http://www.moock.org/asdg/technotes*

Chapter 16, *ActionScript Authoring Environment*

- Revised the section on legacy Smart Clips to cover new Flash MX Components architecture instead

Chapter 17, *Building a Flash Form*

- Revised the code example and tutorial to use *LoadVars* class instead of *loadVariables( )*

Redistributed old Chapter 18, *On-Screen Text Fields* (in first edition only)

- Contents of the entire chapter moved to the *Language Reference* (under *TextField* class) and to Appendix E (and augmented with substantial additions to the *TextField* class)

Removed old Chapter 19, *Debugging* (in first edition only)

- Entire chapter moved to online technote at *http://www.moock.org/asdg/technotes*

Part II, *Language Reference* (formerly Part III)

- Earlier in this Preface, we highlighted the major changes and additions to the *Language Reference*. For a complete list of new methods, properties, classes, objects, global functions, and directives added to the *Language Reference*, see *http://www.moock.org/webdesign/lectures/newInMX*. (Note that *CustomActions* and *LivePreview* are not included in the *Language Reference*, as discussed next.)

What's Not in This Book

Although this book is vast, ActionScript is vaster. It is no longer feasible to cover every possible ActionScript topic within the confines of a single book. We made a conscious editorial decision in this edition to omit formal coverage of the following items (though these topics are covered in passing where relevant):

- Features used exclusively to extend the Flash MX authoring tool (e.g., *CustomActions* and *LivePreview*). These topics are covered in Macromedia's online article "Creating Components in Flash MX" at *http://www.macromedia.com/support/flash/applications/creating_comps*.
- Macromedia's library of Flash UI Components, which extend the authoring tool beyond the core language. See Appendix G, *Flash UI Component Summary*, for a summary of Flash UI Components properties and methods. For resources that cover Flash UI Components in depth, see "Summary" in Chapter 14.
- The Macromedia Flash Communication Server MX (Comm Server) API (e.g., *Remote SharedObject*, *Camera*, *Microphone*, *NetConnection*, and *NetStream*). Comm Server is used to create multiuser web applications with audio and video. See *http://www.macromedia.com/software/flashcom/* for details.
- The basics of the Flash MX authoring tool. However, if you are a programmer who is new to Flash, we give you enough hints so you can input the code examples and follow along. To learn Flash MX animation and graphic design, start with the online help and manual; then explore the web sites listed at *http://www.moock.org/moockmarks*.

There is no CD in the back of the book, but all the code examples can be downloaded from the online Code Depot (cited later in this Preface).

Undocumented ActionScript Features

The Flash development community has a knack for unearthing so-called *undocumented features* of ActionScript—internal abilities of the language that are not officially released or sanctioned for use by Macromedia. In general, use of undocumented features is not recommended because:

- They are not tested for external use and may therefore contain bugs or be unstable.
- They may be removed from future versions of the language without warning.

In this book, we chose to focus on providing the best possible documentation for features that are supported but which may be poorly documented or misdocumented. Therefore, wholly undocumented or unsupported features are not covered unless:

- Macromedia sources have supplied or confirmed the information directly; or
- Use of the feature is so widespread that it demands discussion.

In either case, descriptions in this book of undocumented features include the appropriate warning label in full view. This book covers the following undocumented features:

- `__proto__` (as used to establish inheritance)
- *ASBroadcaster* (partial coverage only, in Chapter 12)
- *ASSetPropFlags()* (partial coverage only, in Chapter 8)
- *LoadVars.decode()*
- *LoadVars.onData()*
- *Object.hasOwnProperty()*
- *System.showSettings()*
- *`TextField`*`.condenseWhite`
- *`TextFormat`*`.font`'s multiple font abilities
- The *XMLNode* class

To see what the ActionScript sleuths have discovered, visit (with prudence):

http://chattyfig.figleaf.com/flashcoders-wiki/index.php?Undocumented%20Features

Flash Naming Conventions

With the introduction of the MX family of products, including Flash MX, Macromedia abandoned a standard numeric versioning system for its Flash authoring tool. The Flash Player, however, is still versioned numerically. Table P-2 describes the naming conventions used in this book for Flash versions.

Table P-2. Flash naming conventions used in this book

Name	Meaning
Flash MX	The Flash MX authoring tool (as opposed to the Flash Player)
Flash Player 6	The Flash Player, version 6. The Flash Player is a browser plugin for major web browsers on Windows and Macintosh. There are both ActiveX and Netscape-style versions of the plugin, but they are referred to collectively as "Flash Player 6" except where noted, such as under *Accessibility* in the *Language Reference*.
Flash Player *x*.0.*y*.0	The Flash Player, specifically, the release specified by *x* and *y*, as in Flash Player 6.0.47.0. See `capabilities.version` in the *Language Reference* for details.
Flash 6	Short for "Flash Player 6," used primarily in the *Language Reference* or wherever the distinction between Flash MX (the authoring tool) and Flash Player 6 (the browser plugin) is irrelevant.
Flash 5 authoring tool	The Flash 5 authoring tool (as opposed to the Flash Player), which came before Flash MX
Flash Player 5	The Flash Player, version 5
Flash 5	Short for "Flash Player 5," used primarily in the *Language Reference* or wherever the distinction between Flash 5 (the authoring tool) and Flash Player 5 (the browser plugin) is irrelevant.
Flash 2, Flash 3, and Flash 4	Versions of the Flash Player prior to version 5, used primarily in the *Language Reference* to indicate which versions of Flash support the given feature.
Standalone Player	A version of the Flash Player that runs directly off the local system, rather than as a web browser plugin or ActiveX control.
Projector	A self-sufficient executable that includes both a *.swf* file and a Standalone Player. Projectors can be built for either the Macintosh or Windows operating system using Flash's File → Publish feature.

What Can ActionScript Do?

ActionScript is used to create all kinds of interactive applications, typically for web-based use. Here are just a few possibilities: an MP3 player, a multiuser drawing application, a 3D walkthrough of a home, an online store, a message board, an HTML editor, and the game Pac-Man. Each of these applications uses a combination of ActionScript's capabilities, a sampling of which follows. Begin thinking about how you can combine these techniques to build your applications.

Timeline Control

Flash movies are composed of frames residing in a linear sequence called the *timeline*. Using ActionScript, we can control the playback of a movie's timeline, play segments of a movie, display a particular frame, halt a movie's playback, loop animations, and synchronize animated content. *Movie clips* within a main movie each have their own timeline.

Interactivity

Flash movies can accept and respond to user input. Using ActionScript, we can create interactive elements such as:

- Buttons that react to mouseclicks (e.g., a classic navigation button)
- GUI elements such as list boxes, combo boxes (a.k.a. drop-down menus), and check boxes
- Content that animates based on mouse movements (e.g., a mouse trailer)
- Objects that can be moved via the mouse or keyboard (e.g., a car in a driving game)
- Text fields that display information on screen or allow users to supply input to a movie (e.g., a fill-in form)

Visual and Audio Content Control

ActionScript can be used to examine or modify the properties of the audio and visual content in a movie. For example, we can change an object's color and location, reduce a sound's volume, or set the font face of a text block. We can also modify these properties repeatedly over time to produce unique behaviors such as animated effects, physics-based motion, and collision detection.

Programmatic Content Generation

Using ActionScript, we can generate visual and audio content directly from a movie's Library or by duplicating existing content on the Stage. In Flash MX, we can use the *MovieClip* class's Drawing API, *createEmptyMovieClip()* method, and *createTextField()* method to create graphics and text from scratch at runtime. Programmatically generated content may serve as a strictly static element—such as a random visual pattern—or as an interactive element—such as a button in a dialog box, an enemy spaceship in a video game, or an option in a pull-down menu.

Server Communication

One of the most common ways to extend Flash's functionality is via communication with some server-side application or script, such as Macromedia ColdFusion MX or a Perl script. Although communicating with ColdFusion is largely the purview of Macromedia Flash Remoting MX (Flash Remoting), the core ActionScript language provides a wide variety of tools for sending information to, and receiving information from, any server-side application or script (e.g., Java, PHP, ASP, etc.). The following applications all involve server communication:

Link to a web page
: See *getURL()*.

Guest book
: See the *LoadVars* and *XML* classes, Chapter 17, and the Code Depot, described in the next section.

Chat application
: See the *XMLSocket* class and the example at *http://www.moock.org/chat*.

Multiplayer networked game
: See the *XMLSocket* class and *http://www.moock.org/unity*.

E-commerce transaction
: See the *LoadVars* and *XML* classes.

Personalized site involving user registration and login
: See the *LoadVars* and *XML* classes.

Detailed implementations of even this limited number of potential ActionScript applications are beyond the scope of this book. Instead, our goal is to give you the fundamental skills to explore the myriad other possibilities on your own. This is not a recipe book—it's a lesson in cooking code from scratch. What's on the menu is up to you.

The Code Depot

We'll encounter dozens of code samples over the upcoming chapters. To obtain relevant source files and many other tutorial files not included in the book, visit the online Code Depot, posted at:

http://www.moock.org/asdg/codedepot

The Code Depot is an evolving resource containing real-world ActionScript applications and code bases. Here's a selected list of samples you'll find in the Code Depot:

- A multiple-choice quiz
- A pan-and-zoom image viewer
- Text field tools, such as an array-to-table converter and a configurable text ticker
- An XML-based chat application
- A guest book application
- A custom mouse pointer and button
- An asteroids game code base
- Programmatic motion effects
- Demos of HTML text fields

- Preloaders
- String manipulation
- Interface widgets, such as slider bars and text scrollers
- Mouse trailers and other visual effects
- Volume and sound control

Additionally, any book news, updates, technotes, and errata will be posted here.

Showcase

Practically every Flash site in existence has at least a little ActionScript in it. But some sites have, shall we say, more than a little. Table P-3 presents a series of destinations that should provide inspiration for your own work. See also the sites listed in Appendix A and the author's bookmarks at *http://www.moock.org/moockmarks*.

Table P-3. ActionScript Showcase

Topic	URL
Experiments in design, interactivity, and scripting	*http://www.yugop.com* *http://www.praystation.com** *http://www.presstube.com* *http://www.pitaru.com* *http://www.flight404.com* *http://www.bzort-12.com* *http://www.benchun.net/mx3d/** *http://www.protocol7.com** *http://www.uncontrol.com** *http://flash.onego.ru** *http://www.figleaf.com/development/flash5** *http://nuthing.com* *http://www.deconcept.com* *http://www.natzke.com*
Games	*http://www.orisinal.com* *http://www.gigablast.com* *http://www.sadisticboxing.com* *http://www.huihui.de* *http://www.sarbakan.com* *http://www.electrotank.com/games/multiuser* *http://www.titoonic.dk/products/games/spider* *http://content.uselab.com/acno* *http://www.neave.com/webgames*

Table P-3. ActionScript Showcase (continued)

Topic	URL
Interface, applications, and dynamic content	*http://www.mnh.si.edu/africanvoices* *http://www.curiousmedia.com* *http://www.smallblueprinter.com* *http://davinci.figleaf.com/davinci* *http://host.oddcast.com* *http://www.enteryourinformation.com/broadmoor/onescreen.cfm*

* Downloadable *.fla* files provided. Otherwise, only *.swf* files available.

Typographical Conventions

In order to indicate the various syntactic components of ActionScript, this book uses the following conventions:

Menu options
: Menu options are shown using the → character, such as File → Open.

`Constant width`
: Indicates code samples, clip instance names, frame labels, property names, and variable names. Variable names often end with the suffixes shown in Table 2-1 (such as `_mc` for variables that refer to movie clip instances). Although using these suffixes is considered the best practice, we sometimes avoided them when we found they made the surrounding text substantially more difficult to read. For brevity, therefore, the preferred suffixes have sometimes been omitted.

Italic
: Indicates function names, method names, class names, layer names, URLs, filenames, and file suffixes such as *.swf*. In addition to being italicized, method and function names are also followed by parentheses, such as *duplicateMovieClip( )*.

`Constant width bold`
: Indicates text that you must enter verbatim when following a step-by-step procedure. **`Constant width bold`** is also used within code examples for emphasis, such as to highlight an important line of code in a larger example.

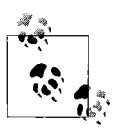
`Constant width italic`
: Indicates code that you must replace with an appropriate value (e.g., *`your name here`*). *`Constant width italic`* is also used to emphasize variable, property, method, and function names referenced in comments within code examples.

In the *Language Reference*, we played around with some font conventions. The following conventions looked the best, while maintaining consistency with our overall approach, so we went for them:

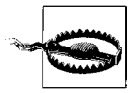
- Class-level properties are shown with both the class name and property in `constant width`, because they should both be entered verbatim, as shown (e.g., `Stage.width`, `Math.NaN`).
- Instance-level properties are shown with the class or object instance in *`constant width italic`*, because the placeholder should be replaced by a specific instance. The property itself is shown in `constant width` and should be entered as shown (e.g., *`Button`*`.tabEnabled`, where *`Button`* should be replaced with a button instance).
- Method and function names, and the class or object to which they pertain, are always shown in italics and followed by parentheses, as in *MovieClip.duplicateMovieClip()*. Refer to the *Language Reference*, surrounding material, and nearby examples to determine whether to include the class name literally, as in *TextField.getFontList()*, or replace it with an instance name, such as *ball_mc.duplicateMovieClip()*.
- Within the *Language Reference*, for brevity, we often omit the class name when discussing a property or method of the class. For example, when discussing the `htmlText` property of the *TextField* class, when we say "set the `htmlText` property," you should infer from context that we mean, "set the *`someField_txt`*`.htmlText` property, where *`someField_txt`* is the identifier for your particular text field."
- In some cases, an object property contains a reference to a method or callback handler. It wasn't always clear whether we should use `constant width` to indicate that it is a property (albeit one storing a method name) or *italics* and parentheses to indicate it is a method (albeit one stored in a property). If the line between a property referring to a method and the method itself is sometimes blurred, forgive us. To constantly harp on the technical difference would have made the text considerably less accessible and readable.
- When summarizing properties for a class, the properties may be shown in *italics*, rather than `constant width`, to save space. This applies only when the properties are summarized under a *Properties* heading and they aren't followed by parentheses, so it is clear that they're properties and not methods.

If any or all of this is confusing now, it will be clear by the time you get to the *Language Reference*, having read about objects, classes, and movie clips in Chapters 12, 13, and 14.

Pay special attention to notes and warnings set apart from the text with the following icons:



This is a tip. It contains useful information about the topic at hand, often highlighting important concepts or best practices.



This is a warning. It helps you solve and avoid annoying problems or warns you of impending doom. Ignore at your own peril.

We'd Like to Hear from You

We have tested and verified the information in this book to the best of our ability, but you may find that features have changed (or even that we have made mistakes!). Please let us know about any errors you find, as well as your suggestions for future editions, by writing to:

O'Reilly & Associates, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472
(800) 998-9938 (in the United States or Canada)
(707) 829-0515 (international/local)
(707) 829-0104 (fax)

We have a web page for the book, where we list errata, examples, or any additional information. You can access this page at:

http://www.oreilly.com/catalog/actscript2

To comment or ask technical questions about this book, send email to:

bookquestions@oreilly.com

For more information about our books, conferences, software, Resource Centers, and the O'Reilly Network, see our web site at:

http://www.oreilly.com

Acknowledgments

As with the first edition, this book would be a mere shadow of itself without the incredible contributions of Macromedia Flash MX's engineering, quality assurance, support, and product management teams. In particular, I can never thank Gary Grossman enough for his critiques, guidance, and patience, not to mention writing the Foreword. Other Macromedians who helped shape this text include: Jonathan Gay, Jeremy Clark, Eric Wittman, Michael Williams, Pete Santangeli, Matt Wobensmith, Ben Chun, Troy Evans, Lee Thomason, Bentley Wolfe, John Dowdell, Rebecca Sun, Janice Pearce, Brian Dister, Henriette Cohn, Jeff Mott, Michael Morris, Deneb Meketa, Tinic Uro, Robert Tatsumi, Colm McKeon, and Mike Chambers.

This book's editor is Bruce Epstein, who I am convinced is superhuman. His knowledge of writing and programming is exceptional, and his ability to bestow that knowledge upon a text is astonishing. I am uncommonly fortunate to be coached by such an outstanding editor (and author in his own right).

Next, it is my honor to present the technical reviewers of this edition, all of whom are members of Macromedia Flash MX's engineering team: Gary Grossman, Chris Thilgen, Gilles Drieu, Nigel Pegg, Slavik Lozben, and Michael Richards. Erica Norton edited the first edition. Thank you, my friends, for your time and devotion.

The beta readers for this edition are all renowned Flash developers for whom I have immense respect: Robert Penner (*http://www.robertpenner.com*), Dave Yang (*http://www.quantumwave.com*), Branden Hall (*http://www.waxpraxis.org*), Amit Pitaru (*http://www.pitaru.com*), Michael Kay (*http://www.peep.org/wizard/*), and Veronique Brossier (*http://www.v-ro.com*). This book's accuracy is in many cases the result of their keen eyes.

Thanks to Tim O'Reilly for setting a standard of thoroughness, quality, and accuracy in everything he publishes. And thanks to O'Reilly's Brian Sawyer, Claire Cloutier, Glenn Bisignani, Mike Sierra, Rob Romano, Edie Freedman, Sandy Torre, and the many copyeditors, indexers, proofreaders, and sales and marketing folks at O'Reilly who helped bring this book to the shelves.

I owe recognition to my good friend Derek Clayton for regularly sharing his programming expertise with me. Derek contributed the Perl code in Chapter 17, the Java *XMLSocket* server in the *Language Reference*, and a generic flat file database system, all available from the online Code Depot. He is also the lead developer of Unity Socket Server, moock.org's commercial application for creating multiuser applications in Flash (*http://www.moock.org/unity*).

To the Flash community: thank you for the inspiration and beauty you create. In particular, thanks to James Patterson, Yugo Nakamura, Naoki Mitsuse, Joshua Davis, James Baker, Marcell Mars, Phillip Torrone, Robert Reinhardt, Mark Fennell, Josh Ulm, Darrel Plant, Todd Purgason, John Nack, Jason Krogh, Hillman Curtis, Glenn Thomas, Hoss Gifford, Manuel Clement, Andreas Heim, Robert Hodgin, Margaret Carlson, Erik Natzke, Andries Odendaal, James Tindall, Jon Williams, Ferry Halim, Jobe Makar, Jared Tarbell, Geoff Stearns, Paul Szypula, Lynda Weinman, the beta readers listed earlier, and whomever I've inevitably omitted.

Many thanks and much love to my wife, Wendy Schaffer, to my parents, and to family and friends. Hopefully this edition wasn't as draining as the first.

And lastly I'd like to thank you, the reader, for taking the time to read this book. I hope it helps to make my passion your own.

—Colin Moock
Toronto, Canada
December 2002

PART I

ActionScript Fundamentals

Part I covers the core syntax and grammar of the ActionScript language: variables, data, statements, functions, event handlers, arrays, objects, movie clips, and components. By the end of Part I, you'll know everything there is to know about writing ActionScript programs.

- Chapter 1, *A Gentle Introduction for Nonprogrammers*
- Chapter 2, *Variables*
- Chapter 3, *Data and Datatypes*
- Chapter 4, *Primitive Datatypes*
- Chapter 5, *Operators*
- Chapter 6, *Statements*
- Chapter 7, *Conditionals*
- Chapter 8, *Loop Statements*
- Chapter 9, *Functions*
- Chapter 10, *Events and Event Handling*
- Chapter 11, *Arrays*
- Chapter 12, *Objects and Classes*
- Chapter 13, *Movie Clips*
- Chapter 14, *Movie Clip Subclasses and Components*
- Chapter 15, *Lexical Structure*
- Chapter 16, *ActionScript Authoring Environment*
- Chapter 17, *Building a Flash Form*

CHAPTER 1

A Gentle Introduction for Nonprogrammers

I'm going to teach you to talk to Flash.

Not just to program in Flash, but to say things to it and listen to what it has to say in return. This is not a metaphor or simply a rhetorical device. It's a philosophical approach to programming.

Programming languages are used to send information to and receive information from computers. They are collections of vocabulary and grammar used to communicate, just like human languages. Using a programming language, we tell a computer what to do or ask it for information. It listens, tries to perform the requested actions, and gives responses. So, while you may think you are reading this book in order to "learn to program," you are actually learning to communicate with Flash. But, of course, Flash doesn't speak English, French, German, or Cantonese. Flash's native language is ActionScript, and you're going to learn to speak it.

Learning to speak a computer language is sometimes considered synonymous with learning to program. But there is more to programming than learning a language's syntax. What would it be like if Flash could speak English—if we didn't need to learn ActionScript in order to communicate with it?

What would happen if we were to say, "Flash, make a ball bounce around the screen?"

Flash couldn't fulfill our request because it doesn't understand the word "ball." Okay, okay, that's just a matter of semantics. What Flash expects us to describe is the objects in the world it knows: movie clips, buttons, frames, and so on. So, let's rephrase our request in terms that Flash recognizes and see what happens: "Flash, make the movie clip named `ball_one` bounce around the screen."

Flash still can't fulfill our request without more information. How big should the ball be? Where should it be placed? In which direction should it begin traveling? How fast should it go? Around which part of the screen should it bounce? For how long? In two dimensions or three? Hmm . . . we weren't expecting all these questions. In reality, Flash doesn't ask us these questions. Instead, when Flash can't understand

us, it just doesn't do what we want it to, or it yields an error message. For now, we'll pretend Flash asked us for more explicit instructions, and reformulate our request as a series of steps:

1. A ball is a circular movie clip symbol named `ball`.
2. A square is a four-sided movie clip symbol named `square`.
3. Make a new green ball 50 pixels in diameter.
4. Call the new ball `ball_one`.
5. Make a new black square 300 pixels wide, and place it in the middle of the Stage.
6. Place `ball_one` somewhere on top of the square.
7. Move `ball_one` in a random direction at 75 pixels per second.
8. If `ball_one` hits one of the sides of the square, make it bounce (reverse course).
9. Continue until I tell you to stop.

Even though we gave our instructions in English, we still had to work through all the logic that governs our bouncing ball in order for Flash to understand us. Obviously, there's more to programming than merely the syntax of programming languages. Just as in English, knowing lots of words doesn't necessarily mean you're a great communicator.

Our hypothetical English-speaking-Flash example exposes four important aspects of programming:

- No matter what the language, the art of programming lies in the formulation of logical steps.
- Before you try to say something in a computer language, it usually helps to say it in English.
- A conversation in one language translated into a different language is still made up of the same basic statements.
- Computers aren't very good at making assumptions. They also have a very limited vocabulary.

Most programming has nothing to do with writing code. Before you write even a single line of ActionScript, think through exactly what you want to do and write out your system's functionality as a flowchart or a blueprint. Once your program has been described sufficiently at the conceptual level, you can translate it into ActionScript.

In programming—as in love, politics, and business—effective communication is the key to success. For Flash to understand your ActionScript, you have to get your syntax absolutely correct, down to the last quote, equals sign, and semicolon. And to assure that Flash knows what you're talking about, you must refer only to the world it knows using terms it recognizes. What may be obvious to you is not obvious to a computer. Think of programming a computer like talking to a child: take nothing for

granted, be explicit in every detail, and list every step that's necessary to complete a task. But remember that, unlike children, Flash will do precisely what you tell it and nothing that you don't.

Some Basic Phrases

On the first day of any language school you'd expect to learn a few basic phrases ("Good day," "How are you," etc.). Even if you're just memorizing a phrase and don't know what each word means, you can learn the effect of the phrase and can repeat it to produce that effect. Once you've learned the rules of grammar, expanded your vocabulary, and used the words from your memorized phrases in multiple contexts, you can understand your early phrases in a richer way. The rest of this chapter will be much like that first day of language school—you'll see bits and pieces of code, and you'll be introduced to some fundamental programming grammar. The rest of the book will build on that foundation. You may want to come back to this chapter when you've finished the book to see just how far you've traveled.

Creating Code

For our first exercise, we'll add four simple lines of code to a Flash movie. Nearly all ActionScript programming takes place in the Actions panel. Any instructions we add to the Actions panel are carried out by Flash when our movie plays. Open the Actions panel now by following these steps:

1. Launch Flash with a new blank document.
2. On the main timeline, select frame 1 of layer 1.
3. Select Window → Actions (F9). (Note that we use the " → " symbol to separate a menu name from a menu option; you should select the "Actions" item from Flash's "Windows" menu.)

The Actions panel is divided into two sections: the Script pane (on the right) and the Toolbox pane (on the left). The Script pane houses all our code. In Normal Mode, the top of the Script pane includes a Parameters pane that simplifies code entry for novices. The Toolbox pane provides us with quick-reference access to the ActionScript language. You'll likely recognize the Movie Control Actions, shown in Figure 1-1, from prior Flash versions.

But there's a lot more to discover in the Toolbox pane. This book covers Operators, Functions, Constants, Properties, and Objects in detail. Some of the Communications components are specific to Macromedia Flash Communication Server MX (Comm Server) and are beyond the scope of this book. Similarly, the Flash UI Components are not part of the core ActionScript language; Appendix G summarizes their properties and methods. Details on the Flash UI Components are available in Flash's documentation, under Help → Tutorials → Introduction to Components.

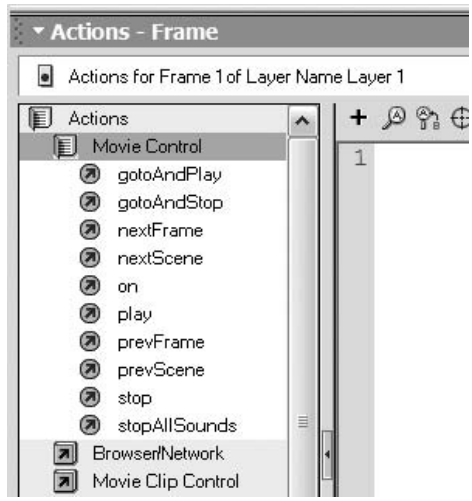


Figure 1-1. Flash MX Movie Control Actions

The Toolbox pane's book-like hierarchical menus can be used to create ActionScript code. However, in order to learn the syntax, principles, and structural makeup of ActionScript, we'll be entering all our code manually.

So-called *Actions* are more than just Actions—they include various fundamental programming-language tools: variables, conditionals, loops, comments, function calls, and so forth. Although many of these are lumped together in one menu, the generic name *Action* obscures the programming structures' significance. We'll be breaking Actions down to give you a programmer's perspective on those structures. Throughout the book, I use the appropriate programming term to describe the Action at hand. For example, instead of writing, "Add a *while* Action," I'll write, "Create a *while* loop." Instead of writing, "Add an *if* Action," I'll write, "Add a new conditional." Instead of writing, "Add a *play* Action," I'll write, "Invoke the *play()* function (or method)." These distinctions are an important part of learning to speak ActionScript.

Ready to get your hands dirty? Let's say hello to Flash.

Say Hi to Flash

Before you can type code into the Actions panel, you must disengage the ActionScript autopilot. From the pop-up menu in the top right corner of the Actions panel, select Expert Mode, as shown in Figure 1-2. Howdya like that? You're already an expert.

For brevity, I'll refer to any pop-up menu in one of Flash's panels as an "Options" menu for the remainder of the book. So I might write, "choose Options → Expert

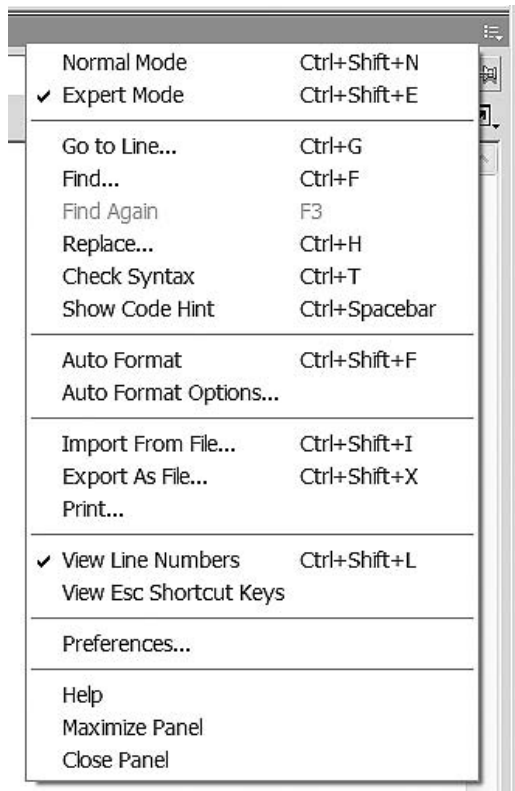


Figure 1-2. Expert Mode selection

Mode in the Actions panel." While you're at it, you can also turn on line numbering using Options → View Line Numbers. See also Chapter 16 for more details on the Actions panel.

When you enter Expert Mode, the Parameters pane disappears from the top of the Actions panel (in Flash 5, the Parameters pane appeared at the bottom of the Actions panel while in Normal Mode). Don't worry—we're not programming with menus, so we won't be needing it.

Next, select frame 1 of layer 1 in the main timeline.

Your ActionScript (a.k.a. *code*) must always be attached to a frame, movie clip, or button; selecting frame 1 causes subsequently created code to be attached to that frame. Flash executes the code attached to a frame when the timeline reaches that frame.

In Expert Mode, you can type directly into the Script pane on the right side of the Actions panel, which is where we'll be doing all our programming.

Here comes the exciting moment—your first line of code. It's time to introduce yourself to Flash. Type the following into the Script pane:

```
var message = "Hi there, Flash!";
```

That line of code constitutes a complete instruction, known as a *statement*. On the line below it, type your second and third lines of code, shown following this paragraph. Replace *your name here* with your first name (whenever you see *constant-width italicized code* in this book it means you have to replace that portion of the code with your own content):

```
var firstName = "your name here";
trace (message);
```

Hmmm. Nothing has happened yet. That's because our code doesn't do anything until we export a *.swf* file and play our movie. Before we do that, let's ask Flash to say hi back to us. Type your fourth line of code under the lines you've already typed (man, we're really on a roll now . . .):

```
trace ("Hi there, " + firstName + ", nice to meet you.");
```

Okay, Flash is ready to meet you. Select Control → Test Movie, and see what happens. Some text should appear in the Output window as shown in Figure 1-3.

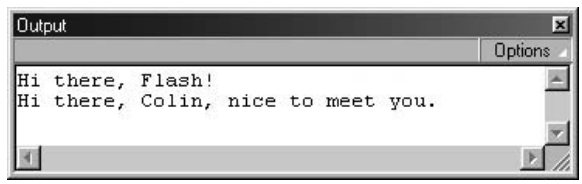


Figure 1-3. Flash gets friendly

Pretty neat, eh? Let's find out how it all happened. To return to editing the *.fla* source file, close the window of the *.swf* file created by our test.

Keeping Track of Things (Variables)

Remember how I said programming was really just communicating with a computer? Well, it is, but perhaps with a little less personality than I've been portraying so far. In your first line of code:

```
var message = "Hi there, Flash!";
```

you didn't really say hi to Flash. You said something more like this:

> Flash, please remember a piece of information for me—specifically, the phrase "Hi there, Flash!" I may need that information in the future, so please give it a label called `message`. If I ask you for `message` later, give me back the text "Hi there, Flash!"

Perhaps not as friendly as saying hi, but it illustrates one of the true foundations of programming: Flash can remember something for you, provided that you label it so that it can be found later. For example, in your second line of code, we had Flash remember your first name, and we named the reference to it `firstName`. Flash remembered your name and displayed it in the Output window, due to the *trace()* command, when you tested your movie.

The fact that Flash can remember things for us is crucial in programming. Flash can remember any type of data, including text (such as your name), numbers (such as 3.14159), and more complex kinds of information that we'll discuss later.

Official variable nomenclature

It's time for a few formal terms to describe how Flash remembers things. So far, you know that Flash remembers data. An individual piece of data is known as a *datum*. A datum (e.g., "Hi there, Flash!") and the label that identifies it (e.g., `message`) are together known as a *variable*. A variable's label is called its *name*, and a variable's datum is called its *value*. We say that the variable *stores* or *contains* its value. Note that "Hi there, Flash!" is surrounded by double quotation marks (quotes) to indicate that it is a *string* of text, not a number or some other kind of information (a.k.a. *datatype*).

In your first line of code, you specified the value of the variable `message`. The act of specifying the value of a variable is known as *assigning the variable's value* or simply *assignment*. But before you can assign a value to a variable, you should first create it (in ActionScript's cousin, JavaScript, you must create variables before using them). We formally bring variables into existence by *declaring* them using the special keyword *var*, which you used earlier.

So, in practice, here's how I might use more formal terms to instruct you to create the first line of code you created earlier: declare a new variable named `message`, and assign it the initial value "Hi there, Flash!" This means you should enter the following code in the Actions panel:

```
var message = "Hi there, Flash!";
```

The Wizard Behind the Curtain (the Interpreter)

Recall your first two lines of code:

```
var message = "Hi there, Flash!";
var firstName = "your name here";
```

In each of those statements, you created a variable and assigned a value to it. Your third and fourth lines, however, are a little different:

```
trace (message);
trace ("Hi there, " + firstName + ", nice to meet you.");
```

These statements use the *trace()* command. You've already seen the effect of that command—it caused Flash to display your text in the Output window. In the third line, Flash displayed the value of the variable `message`. In the last line, Flash also converted the variable `firstName` to its value (whatever you entered as your name) and stuck that into the sentence after the words "Hi there,". The *trace()* command, then, causes any specified data to appear in the Output window (which makes it handy for determining what's going on when a program is running).

The question is, what made the *trace()* command place your text in the Output window? When you create a variable or issue a command, you're actually addressing the *ActionScript interpreter*, which runs your programs, manages your code, listens for instructions, performs any ActionScript commands, executes your statements, stores your data, sends you information, calculates values, and even starts up the basic programming environment when a movie is loaded into the Flash Player.

The interpreter translates your ActionScript into a language that the computer understands and can use to carry out your instructions. During movie playback, the interpreter is always active, dutifully attempting to understand commands you give it. If the interpreter can understand your commands, it sends them to the computer's processor for execution. If a command generates a result, the interpreter provides that response to you. If the interpreter can't understand the command, it either sends you an error message or fails silently. The interpreter, hence, acts like a magic genie—it carries out the orders you specify in your code and reports back to you from Flash when it's done. Like a genie, it always does exactly what you say, not necessarily what you think you mean!

Let's take a closer look at how the interpreter works by examining how it handles a simple *trace()* action.

Consider this command as the interpreter would:

```
trace ("Nice night to learn ActionScript.");
```

The interpreter immediately recognizes the keyword *trace* from its special list of legal command names. The interpreter also knows that *trace()* is used to display text in the Output window, so it also expects to be told what text to display. It finds, "Nice night to learn ActionScript." between parentheses following the word *trace* and thinks, "Aha! That's just what I need. I'll have that sent to the Output window right away!"

Note that the entire command is terminated by a semicolon (`;`). The semicolon acts like the period at the end of a sentence; with few exceptions, every ActionScript statement should end with a semicolon. With the statement successfully understood and all the required information in hand, the interpreter translates the command for the processor to execute, causing our text to appear in the Output window.

That's a gross oversimplification of the internal details of how a computer processor and an interpreter work, but it illustrates these points:

- The interpreter is always listening for your instructions.
- The interpreter has to read your code, letter by letter, and try to understand it. This is the same as you trying to read and understand a sentence in a book.
- The interpreter reads your ActionScript using strict rules—if the parentheses in our *trace()* statement were missing, for example, the interpreter wouldn't be able to understand what's going on, and the command would fail.

You've only just been introduced to the interpreter, but you'll be as intimate with it as you are with a lover before too long: lots of fights, lots of yelling—"Why aren't you listening to me?!"—and lots of beautiful moments when you understand each other perfectly. Strangely enough, my dad always told me the best way to learn a new language is to find a lover that speaks it. May I, therefore, be the first to wish you all the best in your new relationship with the ActionScript interpreter. From now on, I'll regularly refer to "the interpreter" interchangeably with "Flash" when describing how ActionScript instructions are carried out.

Extra Info Required (Arguments)

You've already seen one case in which we provided the interpreter with the text to display when issuing a *trace()* command. This approach is common; we'll often provide the interpreter with ancillary data required to execute a command. There's a special name for a datum sent to a command: an *argument*, or synonymously, a *parameter*. To supply an argument to a command, enclose the argument in parentheses, like this:

```
command(argument);
```

When supplying multiple arguments to a command, separate them with commas, like this:

```
command(argument1, argument2, argument3);
```

Supplying an argument to a command is known as *passing* the argument. For example, in the code phrase *gotoAndPlay(5)*, the word *gotoAndPlay* is the name of the command, and *5* is the argument being passed (in this case the frame number). Some commands, such as *stop()*, require parentheses but do not accept arguments. You'll learn why in Chapter 9, where we discuss functions in detail.

ActionScript's Glue (Operators)

Let's take another look at your fourth line of code, which contains this *trace()* statement:

```
trace ("Hi there, " + firstName + ", nice to meet you.");
```

See the + (plus) signs? They're used to join (*concatenate*) our text together and are but one of many available *operators*. The operators of a programming language are akin to conjunctions ("and," "or," "but," etc.) in human languages. They're devices

used to combine and manipulate phrases of code. In the *trace()* example, the plus operator joins the quoted text "Hi there, " to the text contained in the variable `firstName`.

All operators link phrases of code together, manipulating those phrases in the process. Whether the phrases are text, numbers, or some other datatype, an operator almost always performs some kind of transformation. Very commonly, operators combine two things together, as the plus operator does. But other operators compare values, assign values, facilitate logical decisions, determine datatypes, create new objects, and provide various other handy services.

When used with two numeric operands, the plus sign (+) and the minus sign (–), perform basic arithmetic. The following command displays "3" in the Output window:

```
trace(5 - 2);
```

The *less-than* operator (<) checks which of two numbers is smaller or determines which of two letters is alphabetically first:

```
if (3 < 300) {
  // Do something...
}
if ("a" < "z") {
  // Do something else...
}
```

The combinations, comparisons, assignments, or other manipulations performed by operators are known as *operations*. Arithmetic operations are the easiest operations to understand, because they follow basic mathematics: addition (+), subtraction (–), multiplication (*), and division (/). But some operators will be less recognizable to you, because they perform specialized programming tasks. Take the *typeof* operator, for example. It tells us what kind of data is stored in a variable. So, if we create a variable `x`, and assign it the value 4, we can then ask the interpreter what datatype `x` contains, like this:

```
var x = 4;
trace (typeof x);
```

When this code is executed, Flash displays the word "number" in the Output window. Notice that we provided the *typeof* operator with a value upon which to operate—`x`—but without using parentheses: *typeof* `x`. You might therefore wonder whether `x` is an *argument* of *typeof*. In fact, `x` plays the same role as an argument (it's an ancillary piece of data needed in the computation of the phrase of code), but in the context of an operator, it is officially called an *operand*. An operand is an item upon which an operator operates. For example, in the expression 4 + 9, the numbers 4 and 9 are operands of the + operator.

Chapter 5 covers ActionScript operators in detail. For now, just remember that operators link together phrases of code in an expression.

Putting It All Together

Let's review what you've learned. Here, again, is line one of our example:

```
var message = "Hi there, Flash!";
```

The keyword *var* tells the interpreter that we're declaring (creating) a new variable. The word `message` is the name of our variable (a name we've chosen arbitrarily). The equals sign is an operator that assigns the text string ("Hi there, Flash!") to the variable named `message`. Hence, the text "Hi there, Flash!" becomes the value of `message`. Finally, the semicolon (`;`) tells the interpreter that we're finished with our first statement.

Line two is pretty much the same as line one:

```
var firstName = "your name here";
```

It assigns the text string you typed in place of *your name here* to the variable `firstName`. A semicolon ends our second statement.

We reuse the variables `message` and `firstName` in lines three and four:

```
trace (message);
trace ("Hi there, " + firstName + ", nice to meet you.");
```

The keyword *trace* tells the interpreter to display some text in the Output window. We pass the text to be displayed as an argument. The opening parenthesis marks the beginning of our argument. The *trace()* command requires one argument, but that argument might be a fairly complex expression. For example, in line four, the argument includes two *operations*, both of which use the plus *operator*, joining three *operands*. The first operation joins its first operand, "Hi there, " to the value of its second operand, `firstName`. The second operation joins the text ", nice to meet you." to the result of the first operation. The closing parenthesis marks the end of our argument, and the semicolon once again terminates our statement. Technically, parentheses demarcate a list of one or more arguments. If a command requires more than one argument, the arguments are separated by commas (the commas in the preceding example are part of the text, because they are enclosed in quotes, and should not be confused with commas used to separate multiple arguments, as shown later).

Blam! You've written your first ActionScript program. That has a nice ring to it, and it's an important landmark.

Further ActionScript Concepts

You've already seen many of the fundamental elements that make up ActionScript: data, variables, operators, statements, functions, and arguments. Before we delve deeper into those topics, let's sketch out the rest of ActionScript's core features.

Flash Programs

To most computer users, a *program* is synonymous with an *application*, such as Adobe Photoshop or Macromedia Dreamweaver MX. Obviously, that's not what we're building when we program in Flash. Programmers, on the other hand, define a program as a collection of code (a series of statements), but that's only part of what we're building.

A Flash movie is more than a series of lines of code. Code in Flash is intermingled with Flash movie elements, such as frames, movie clips, graphics, and buttons. In the end, there really isn't such a thing as a Flash "program" in the classic sense of the term. Instead of complete programs written in ActionScript, we have *scripts*: code segments that give programmatic behavior to our movie, just as JavaScript scripts give programmatic behavior to HTML documents. The real product we're building is not a program but a complete Flash movie (the exported *.swf* file, including its code, timelines, visuals, sound, and other assets).

Our scripts include most of what you'd see in traditional programs, without the operating system–level stuff you would write in languages like C++ or Java to place graphics on the screen or cue sounds. We're spared the need to manage the nuts and bolts of graphics and sound programming, which allows us to focus most of our effort on designing the behavior of our movies.

Expressions

The statements of a script, as we've learned, contain the script's instructions. But most instructions are pretty useless without data. When we set a variable, for example, we assign some data as its value. When we use the *trace()* command, we pass data as an argument for display in the Output window. Data is the content we manipulate in our ActionScript code. Throughout your scripts, you'll retrieve, assign, store, and generally sling around a lot of data.

In a program, any phrase of code that yields a single datum when a program runs is referred to as an *expression*. The number 7 and the string "Welcome to my web site," are both very simple expressions. They represent simple data that will be used as-is when the program runs. As such, these expressions are called *literal expressions*, or *literals* for short.

Literals are only one kind of expression. A variable may also be an expression (variables contain data and can stand in wherever data is needed, so they count as expressions). Expressions get even more interesting when they are combined with operators to form larger expressions. For example, the complex expression *4 + 5* includes two operands, 4 and 5, but the plus operator makes the entire expression yield the single value 9. Complex expressions may contain other, shorter expressions, provided that the entire phrase of code can still be converted into a single value.

Here we see the variable message:

```
var message = "Hi there, Flash!";
```

If we like, we can combine the variable expression message with the literal expression " How are you?" as follows:

```
message + " How are you?"
```

which becomes "Hi there, Flash! How are you?" when the program runs. You'll frequently see complex expressions that include shorter expressions when working with arithmetic, such as:

```
(2 + 3) * (4 / 2.5) - 1
```

It's important to be exposed to expressions early in your programming career, because the term "expression" is often used in descriptions of programming concepts. For example, I might write, "To assign a value to a variable, type the name of the variable, then an equals sign followed by any expression."

Two Vital Statement Types: Conditionals and Loops

In nearly all programs, we'll use *conditionals* to add logic to our programs and *loops* to perform repetitive tasks.

Making choices using conditionals

One of the really rewarding aspects of Flash programming is making your movies smart. Here's what I mean by smart: Suppose a girl named Wendy doesn't like getting her clothes wet. Before Wendy leaves her house every morning, she looks out the window to check the weather, and if it's raining, she brings an umbrella. Wendy's smart. She uses basic logic—the ability to look at a series of options and make a decision about what to do based on the circumstances. We use the same basic logic when creating interactive Flash movies.

Here are a few examples of logic in a Flash movie:

- Suppose we have three sections in a movie. When a user goes to each section, we use logic to decide whether to show her the introduction to that section. If she has been to the section before, we skip the introduction. Otherwise, we show the introduction.
- Suppose we have a section of a movie that is restricted. To enter the restricted zone, the user must enter a password. If the user enters the right password, we show her the restricted content. Otherwise, we prohibit access.
- Suppose we're moving a ball across the screen, and we want it to bounce off a wall. If the ball reaches a certain point, we reverse the ball's direction. Otherwise, we let the ball continue traveling in the direction it was going.

These examples of movie logic require a special type of statement called a *conditional*. Conditionals let us specify the terms under which a section of code should—or should not—be executed. Here's an example of a conditional statement:

```
if (userName == "James Bond") {
  trace ("Welcome to my web site, 007.");
}
```

The generic structure of a conditional is:

```
if (this condition is met) {
  then execute these lines of code
}
```

You'll learn more about the detailed syntax of conditionals in Chapter 7. For now, remember that a conditional allows Flash to make logical decisions.

Repeating tasks using loops

Not only do we want our movies to make decisions, we want them to do tedious, repetitive tasks for us. That is, until they take over the world and enslave us and grow us in little energy pods as . . . wait . . . forget I told you that . . . ahem. Suppose you want to display a sequence of five numbers in the Output window, and you want the sequence to start at a certain number. If the starting number is 1, you can display the sequence like this:

```
trace (1);
trace (2);
trace (3);
trace (4);
trace (5);
```

But if you want to start the sequence at 501, you'd have to retype all the numbers as follows:

```
trace (501);
trace (502);
trace (503);
trace (504);
trace (505);
```

We can avoid that retyping by making our *trace()* statements depend on a variable, like this:

```
var x = 1;
trace (x);
x = x + 1;
trace (x);
x = x + 1;
trace (x);
x = x + 1;
trace (x);
x = x + 1;
trace (x);
```

On line 1, we set the value of the variable x to 1. Then at line 2, we display that value in the Output window. On line 3, we say, "Take the current value of x, add 1 to it, and stick the result back into our variable x," so x becomes 2. Then we display the value of x in the Output window again. We repeat this process three more times. By the time we're done, we've displayed a sequence of five numbers in the Output window. The beauty is that if we now want to change the starting number of our sequence to 501, we just change the initial value of x from 1 to 501. Because the rest of our code is based on x, the entire sequence changes when we change the initial value.

That's an improvement over our first approach, and it works pretty well when we're displaying only five numbers, but it becomes impractical if we want to count to 1,000. To perform highly repetitive tasks, we use a *loop*—a statement that causes a block of code (i.e., one or more instructions) to be performed a specified number of times. There are several types of loops, each with its own syntax. One of the most common loop types is the *while* loop. Here's what our counting example would look like as a *while* loop instead of as a series of individual statements:

```
var x = 1;
while (x <= 5) {
  trace (x);
  x = x + 1;
}
```

The keyword *while* indicates that we want to start a loop. The expression *(x <= 5)* governs how many times the loop should execute (as long as x is less than or equal to 5), and the statements *trace(x);* and *x = x + 1;* are executed with each repetition (*iteration*) of the loop. As it is, our loop saves us only five lines of code, but it could save us hundreds of lines if we were counting to higher numbers. And our loop is flexible. To make our loop count to 1,000, we simply change the expression *(x <=5)* to *(x <=1000)*:

```
var x = 1;
while (x <= 1000) {
  trace (x);
  x = x + 1;
}
```

Like conditionals, loops are one of the most frequently used and important types of statements in programming.

Modular Code (Functions)

So far, your longest script has consisted of five lines of code. But it won't be long before those five lines become 500, or maybe even 5,000. When you want to manage your code more efficiently, reduce your work, and make your code easier to apply to multiple scenarios, you'll learn to love *functions*. A function is a packaged series of

statements that performs some useful action or calculation. In practice, functions serve mostly as reusable blocks of code.

Suppose you want to write a script that calculates the area of a 4-sided figure. Without functions, your script might look like this:

```
var height = 10;
var width = 15;
var area = height * width;
```

Now suppose you want to calculate the area of five 4-sided figures. Your code quintuples in length:

```
var height1 = 10;
var width1 = 15;
var area1 = height1 * width1;
var height2 = 11;
var width2 = 16;
var area2 = height2 * width2;
var height3 = 12;
var width3 = 17;
var area3 = height3 * width3;
var height4 = 13;
var width4 = 18;
var area4 = height4 * width4;
var height5 = 20;
var width5 = 5;
var area5 = height5 * width5;
```

Because we're repeating the area calculation over and over, we are better off putting it in a function once and executing that function multiple times:

```
function area(height, width){
  return height * width;
}
area1 = area(10, 15);
area2 = area(11, 16);
area3 = area(12, 17);
area4 = area(13, 18);
area5 = area(20, 5);
```

In this example, we first create the area-calculating function using the *function* statement, which defines (declares) a function just as *var* declares a variable. Then we give our function a name, `area`, just as we give variables names. Between the parentheses, we list the arguments that our function receives every time it's used: `height` and `width`. And between the curly braces ({ }), we include the statement we want executed:

```
return height * width;
```

After we create a function, we may run the code it contains from anywhere in our movie by using the function's name. In our example we called the *area()* function five times, passing it the `height` and `width` values it expects each time: *area(10, 15)*, *area(11, 16)*, and so on. The result of each calculation is returned to us, and we store those results in the variables `area1` through `area5`. This is nice and neat, and much

less work than the nonfunction version of our code shown earlier. Enclosing reusable code within functions save us time and reduces the number of lines of code needed. It also makes our code much more legible and modular. Someone reading the code can probably guess what the *area()* function does, based solely on its name.

Don't fret if you have questions about this function example; we'll learn more about functions in Chapter 9. For now, just remember that functions give us an extremely powerful way to create complex systems. Functions help us reuse our code and package its functionality, extending the limits of what is practical to build.

Built-in functions

Notice that functions take arguments just as the *trace()* command does. Invoking the function *area(4, 5);* looks very much the same as issuing the *trace()* command such as *trace (x);* except that the *area()* function requires two arguments, which are separated by commas, whereas the *trace()* command requires a single argument. The similarity is not a coincidence. As we learned earlier, many Actions, including the *trace()* Action, are actually functions. But they are a special type of function that is built into ActionScript (as opposed to user-defined functions, like our *area()* function). It is, therefore, legitimate—and technically more accurate—to say, "Call the *gotoAndStop()* function," than to say, "Execute a *gotoAndStop* Action." A built-in function is simply a reusable block of code that comes with ActionScript for our convenience. Built-in functions let us do everything from performing mathematical calculations to controlling movie clips. All the built-in functions are listed in Part II, the *Language Reference*. We'll encounter many of them as we learn ActionScript's fundamentals.

Objects and Object-Oriented Programming

We've learned that a *statement* is an ActionScript command that makes Flash do something. And we know that a *function* groups multiple statements into a single convenient command. Functions provide organization for our code that lets us build more complex programs. *Objects* take the next logical step—they package a series of related functions and variables into a single code module.

As a program's complexity increases, it becomes more and more difficult to describe its operation as a single linear set of instructions (statements). Objects help us conceptually model the behavior of a program as an interrelated series of self-contained components, rather than a long list of commands. For example, in the bouncing ball example from the beginning of this chapter, we could use objects to represent the bouncing ball and the square room in which the ball bounces. The ball object would encompass functions that relate to the ball, such as *startMoving()*, *stopMoving()*, and *checkForWall()*. It would also contain information about the ball, stored in variables such as `velocity`, `color`, and `diameter`. The room object might include the functions *setRoomSize()* and *addBall()* and the variables `width` and `height`. When we use

objects, our program's conceptual model is reflected by an intuitive syntax that looks like this:

```
room.addBall( );
ball.diameter = 5;
ball.startMoving( );
```

In general terms, the object (noun) comes first, followed by a dot (.), followed by either a function (verb) or a variable (adjective). The function does something to the object, while the variable tells us something about the object. Functions associated with an object are often called *methods*. Variables associated with an object are often called *properties*. Here are some examples:

```
object.method( );
boy.run( );
someMovieClip.play( );

object.property = value;
boy.speed = 5;
someMovieClip._width = 60;
```

Object-oriented programming (OOP) is an approach in which objects are the fundamental building blocks of a program. There are, however, varying degrees of OOP depending on the language and the developer's taste. Purists insist that every part of a program should be contained by an object (this rule is enforced by the Java programming language). Others are happy using objects to structure only certain parts of a program (this is common in ActionScript).

Regardless, you should become familiar with objects, because they are intrinsic to ActionScript's makeup. Nearly everything in Flash—from buttons to text fields to movie clips—is represented in ActionScript by an object. Even if you never organize your own code with objects, you'll have to use object-oriented programming techniques to control the buttons, text, and movie clips in your movie. Chapter 12 covers object-oriented programming in exhaustive detail. For now, just start thinking of things in Flash, such as movie clips, as objects that can perform certain tasks (e.g., *play( )*) or store information (e.g., `_width`).

Movie Clip Instances

With all this talk about programming fundamentals, I hope you haven't forgotten about the basics of Flash. One of the keys to visual programming in Flash is movie clip *instances*. As a Flash designer or developer, you should already be familiar with movie clips, but you may not think of movie clips as programming devices.

Every movie clip has a symbol definition that resides in the Library of a Flash movie. We can add many copies, or *instances*, of a single movie clip symbol to a Flash movie by dragging the clip from the Library onto the Stage. We can also create movie clips at runtime via ActionScript. All visual programming in Flash involves some degree of movie clip instance control. For example, a bouncing ball is nothing more than a

movie clip instance that is repositioned on the Stage repetitively. Using ActionScript, we can alter an instance's location, size, current frame, rotation, and so forth, during the playback of our movie.

If you're unfamiliar with movie clips and instances, see "Working with Movie Clips and Buttons" under Help → Using Flash before continuing with the rest of this book.

The Event-Based Execution Model

The final topic in our overview of ActionScript fundamentals is the *execution model*, which dictates when the code in your movie runs (is executed). You may have code attached to various frames, buttons, and movie clips throughout your movie, but when does it all actually run? To answer that question, let's take a short stroll down computing history's memory lane.

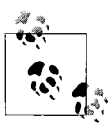
In the early days of computing, a program's instructions were executed sequentially, in the order that they appeared, starting with the first line and ending with the last line. The program was meant to perform some action and then stop. That kind of linear program, called a *batch* program, doesn't handle the interactivity required of an *event-based* programming environment like Flash.

Event-based programs don't run in a linear fashion. They run continuously (in an *event loop*), waiting for things (*events*) to happen and executing code segments in response to those events. In a language designed for use with a visual interactive environment (such as ActionScript or JavaScript), the events are typically user actions such as mouseclicks or keystrokes.

When an event occurs, the interpreter notifies your program of the event. Your program can then react to the event by asking the interpreter to execute an appropriate segment of code. For example, if a user clicks a button in a movie, we could execute some code that displays a different section of the movie (navigation) or submits variables to a database (form submission).

But programs don't react to events unless we create *event handlers* and/or *event listeners* that tell the interpreter which function to run when a certain event happens. In Flash MX, we create an event handler using the general form:

```
someObject.onSomeEvent = someFunction;
```



Flash MX allows this event syntax for movie clips and buttons, whereas Flash 5 required a special event syntax for movie clips and buttons. For details, see Chapter 10.

Events always occur in relation to some object in our program. For example, a button might rotate the movie clip in which it resides using the following event handler:

```
rotateButton.onRelease = rotate;
```

In natural language, this tells the interpreter, "When the mouse button is released over the rotateButton object, execute the *rotate* function." The *rotate* function might look like this:

```
function rotate ( ) {
  this._parent._rotation = 45;
}
```

A function that is executed when an event occurs is known as a *callback function*. As we'll learn in Chapter 10, within a callback function, the keyword this refers to the object that defined the event handler (in our case, rotateButton). In the case of a button reacting to a mouseclick, this refers to the button that was clicked. Using object-oriented syntax, the movie clip in which the button resides is referred to as this._parent (the movie clip is the button's _parent because it contains the button). Finally, we set the rotation of the parent movie clip to 45 degrees by assigning 45 to this._parent._rotation:

```
this._parent._rotation = 45;
```

This literally translates to, "Set the rotation of this button's parent movie clip to 45 degrees."

Our sample button event handler is commonly written more succinctly as:

```
rotateButton.onRelease = function ( ) {
  this._parent._rotation = 45;
};
```

Event-based programs are always running an event loop, ready to react to the next event. Events are crucial to interactivity. Without events, our scripts wouldn't do anything—with one exception: Flash executes any code on a frame when the playhead enters that frame. The implied event is simply the playhead entering the particular frame, which is so intrinsic to Flash that no explicit event handler is required.

Events literally make things happen, which is why they come at the end of your first day of ActionScript language school. You've learned what's involved in writing scripts and that events govern when those scripts will be executed. I'd say you're ready to try your first real conversation.

Building a Multiple-Choice Quiz

Now that we've explored the basic principles of ActionScript, let's apply those principles in the context of a real Flash movie. We'll start our applied study of Flash programming by creating a multiple-choice quiz, using very simple programming techniques, most of which you've already learned. We'll revisit our quiz in later chapters to see how it can be improved after learning more advanced programming concepts. We'll eventually make the code more elegant so that it's easier to extend and maintain, and we'll add more features to our quiz so that it can easily handle any number of questions.

The finished *.fla* file for this quiz can be found in the online Code Depot, cited in the Preface. This is a lesson in Flash programming, not Flash production. I assume that you are already comfortable creating and using buttons, layers, frames, keyframes, and the Text tool. If not, consult the Flash Help documentation. The quiz shows a real-world application of the following aspects of ActionScript programming:

- Variables
- Controlling the playhead of a movie with functions
- Button event handlers
- Simple conditionals
- Text field objects used for on-screen display of information

Quiz Overview

Our quiz, part of which is shown in Figure 1-4, will have only two questions. Each question comes with three multiple-choice answers. Users submit their answers by clicking the button that corresponds to their desired selections. The selections are recorded in a variable that is later used to grade the quiz. When all the questions have been answered, the number of correct answers is tallied, and the user's score is displayed.

Building the Layer Structure

When building Flash movies, it's important to organize your content into manageable divisions by keeping different content elements on individual layers. Layering content is a good production technique in general, but it is essential in Flash programming. In our quiz, and in the vast majority of our scripted movies, we'll keep all our timeline scripts on a single isolated layer, called *scripts*. I keep the *scripts* layer as the first one in my layer stack so that it's easy to find.

We'll also keep all our frame labels on a separate layer, called (surprise, surprise) *labels*. The *labels* layer should live beneath the *scripts* layer on all your timelines. In addition to these two standard layers (*scripts* and *labels*), our quiz movie has a series of content layers on which we'll isolate our various content assets.

Start building your quiz by creating and naming the following layers (using Insert → Layer) and arranging them in this order:

scripts
labels
quiz end
question 2
question 1
choice buttons
housing

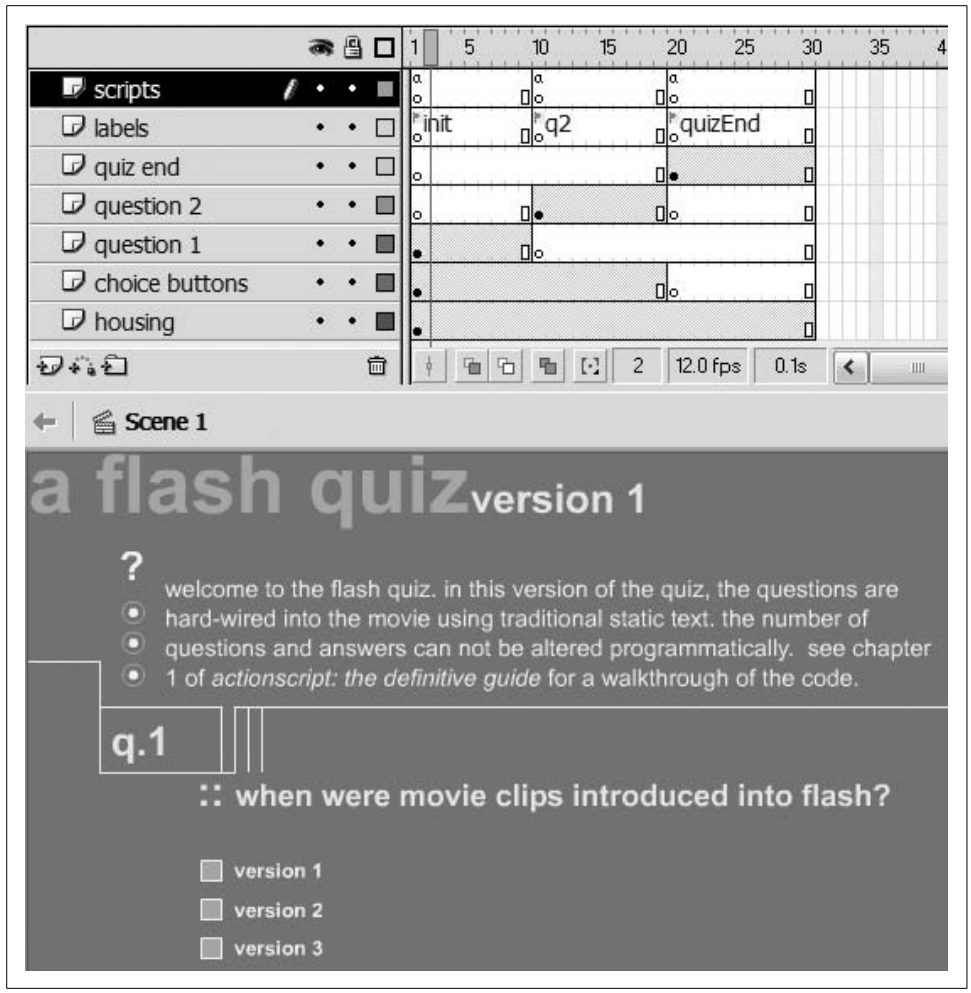


Figure 1-4. A Flash quiz

Now add 30 frames to each of your layers (by highlighting 30 frames in the timeline and choosing Insert → Frame or F5). Your timeline should look like the one in Figure 1-5.

Creating the Interface and Questions

Before we get to the scripts that run the quiz, we need to set up the questions and the interface that will let the user progress through the quiz.

Follow these steps to create the questions and quiz title:

1. With frame 1 of the *housing* layer selected, use the Text tool to type your quiz title directly on the Stage.

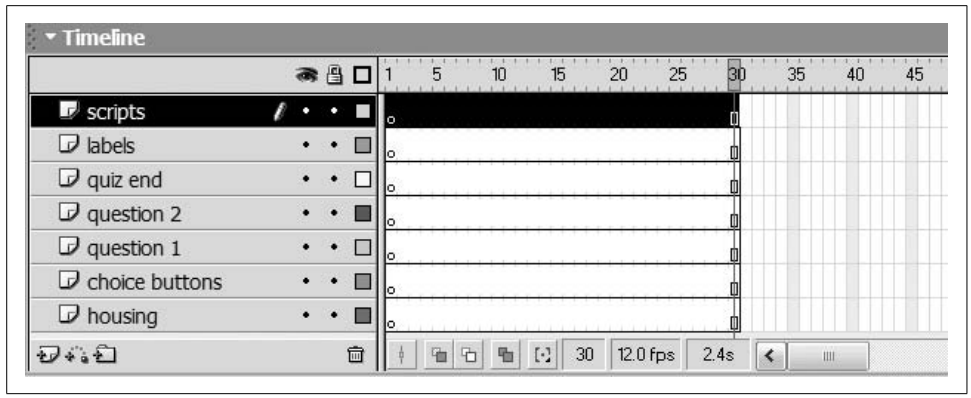


Figure 1-5. Quiz timeline initial setup

2. At frame 1 of the *question 1* layer, add the question number "1" and the text for Question 1, "When were movie clips introduced into Flash?" Leave room for the answer text and buttons below your question.
3. Below your question text (still on the *question 1* layer), add the text of your three multiple-choice answers: "Version 1," "Version 2," and "Version 3," each on its own line.
4. We'll use Question 1 as a template for Question 2. Select the first frame of the *question 1* layer and choose Edit → Copy Frames (not Edit → Copy).
5. Select frame 10 of the *question 2* layer and choose Edit → Paste Frames (not Edit → Paste). A duplicate of your first question appears on the *question 2* layer at frame 10.
6. To prevent Question 1 from appearing behind Question 2, add a blank keyframe at frame 10 of the *question 1* layer using Insert → Blank Keyframe.
7. Back on frame 10 of the *question 2* layer, change the question number from "1" to "2" and change the text of the question to, "When was MP3 audio support added to Flash?" Change the multiple-choice answers to "Version 3," "Version 4," and "Version 5."

Our questions are almost complete, but we must add the buttons the user will press to answer each question:

1. Create a simple button symbol (Insert → New Symbol → Button) that looks like a checkbox or radio button and measures no higher than a line of text (see the buttons in Figure 1-4).
2. At frame 1 of the *choice buttons* layer, next to your three Question 1 answers, place three instances of your checkbox button.
3. Select the topmost button (next to the first answer), and, using the Property inspector, set the <Instance Name> to `choice1_btn`.
4. Repeat step 3 to name the remaining two answer buttons `choice2_btn` and `choice3_btn` (from top to bottom).

Figure 1-6 shows how your timeline will look after you've added the two questions to the quiz. Notice that we use frames on the timeline to represent so-called *application states*. Each "screen" of the quiz application gets its own frame. Later we'll add labels to the question frames and see how this facilitates navigation between frames within our program, allowing us to display the desired state—in this case, the appropriate question—easily.

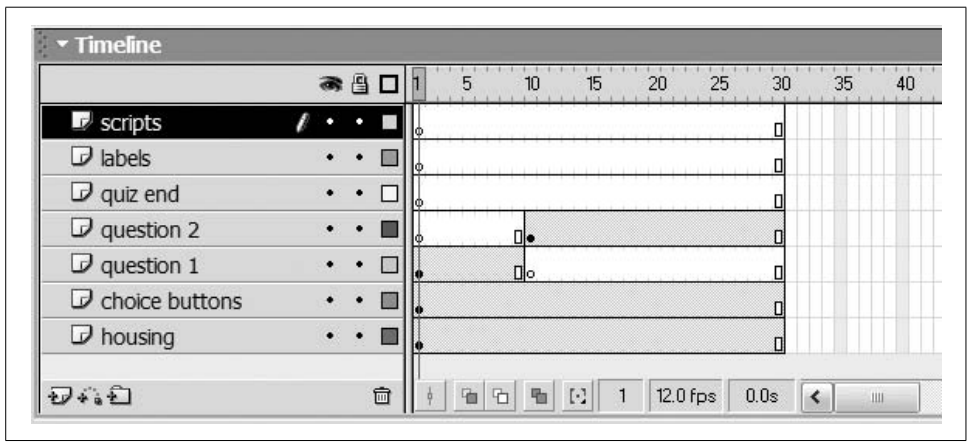


Figure 1-6. Quiz timeline with two questions

Initializing the Quiz

Our first order of business in our quiz script is to create the variables we'll use throughout our movie.

In our quiz, we create variables on the first frame of the movie, but in other movies we'll normally do it after preloading part or all of the movie. Either way, we want to initialize our variables before any other scripting occurs.

Once our variables are defined, we invoke the *stop()* function to keep the user paused on the first frame (where the quiz starts).

For more complex situations, we may also set the initial state of our application by calling functions in preparation for the rest of the movie. This step is known as *initialization*. A function that starts processes in motion or defines the initial conditions under which a system operates is traditionally named *init*.

Before attaching the quiz's *init* code, shown in Example 1-1, to frame 1 of the *scripts* layer, let's examine what it does.

Example 1-1. Init code for quiz

```
// Init main timeline variables
var q1answer;          // User's answer for question 1
var q2answer;          // User's answer for question 2
var totalCorrect = 0;  // Counts number of correct answers

// Stop the movie at the first question
stop();
```

Line 1 of our *init* sequence is a *comment*. Comments are notes that you add to your code to explain what's going on. A single-line comment starts with two forward slashes and (preferably) a space, which is then followed by a line of text:

```
// This is a comment
```

Notice that comments can be placed at the end of a line, following your code, like this:

```
var x = 5; // This is also a comment
```

Line 2 of Example 1-1 creates a variable named `q1answer`. Recall that to create a variable we use the *var* keyword followed by a variable name, as in:

```
var favoriteColor;
```

So, the second through fourth lines of our code declare the variables we'll need, complete with comments explaining their purpose:

- `q1answer` and `q2answer` will contain the value of the user's answer (1, 2, or 3, indicating which of the three multiple-choice answers was selected for each question). We'll use these values to check whether the user answered the questions correctly.
- `totalCorrect` will be used at the end of the quiz to tally the number of questions that the user answered correctly.

Take a closer look at Line 4 of Example 1-1:

```
var totalCorrect = 0;  // Counts number of correct answers
```

Line 4 performs double duty; it first declares the variable `totalCorrect` and then assigns the value 0 to that variable using the assignment operator, `=`. We initialize `totalCorrect` to 0 because the user hasn't answered any of the questions correctly at the beginning of the quiz. The other variables don't need default values because they are set explicitly during the quiz.

After our variables have been defined, we call the *stop()* function, which halts the playback of the movie at the current frame—in this case, frame 1—where the quiz begins:

```
// Stop the movie at the first question
stop();
```

Observe, again, the use of the comment before the *stop()* function call. That comment explains the intended effect of the code that follows.

Comments are optional, but they help clarify our code in case we leave it for a while and need a refresher when we return, or if we pass our code onto another developer. Comments also make code easy to scan, which is important during debugging.

Now that you know what our *init* code does, let's attach it to frame 1 of our quiz movie's *scripts* layer:

1. Select frame 1 of the *scripts* layer.
2. Use Window → Actions (F9) to open the Actions panel.
3. Make sure you're using Expert Mode, which can be set as a permanent preference via the pop-up Options menu in the top right corner of the Actions Panel.
4. In the right side of the Actions panel, type the *init* code as shown earlier in Example 1-1.

Adding Frame Labels

We've completed our quiz's *init* script and built our questions. We will now add some frame labels to help control the playback of our quiz.

In order to step the user through our quiz one question at a time, we've separated the content for Question 1 and Question 2 into frames 1 and 10. By moving the playhead to those keyframes, we'll create a slide show effect, where each slide contains a question. We know that Question 2 is on frame 10, so when we want to display Question 2, we can call the *gotoAndStop()* function like this:

```
gotoAndStop(10);
```

which causes the playhead to advance to frame 10, the location of Question 2. A sensible piece of code, right? Wrong! Whereas using the specific number 10 with our *gotoAndStop()* function works, it isn't flexible (using literals in this manner is called *hardcoding*, which is often ill-advised). If, for example, we added five frames to the timeline before frame 10, Question 2 would suddenly reside at frame 15, and our *gotoAndStop(10)* command would not bring the user to the correct frame. To allow our code to work even if the frames in our timeline shift, we use *frame labels* instead of frame numbers. Frame labels are expressive names, such as `q2` or `quizEnd`, attached to specific points on the timeline.

Once a frame is labeled, we can use the label to refer to the frame by name instead of by number. The flexibility of frame labels is indispensable. I hardly ever use frame numbers with playback-control functions like *gotoAndStop()*.

Variable Naming Styles

By now you've seen quite a few variable names, and you may be wondering about the capitalization. If you've never programmed before, a capital letter in the middle of a word, as in `firstName`, or `totalCorrect`, may seem odd. Capitalizing the second word (and any subsequent words) of a variable name visually demarcates the words within that name. We use this technique because spaces and dashes aren't allowed in a variable name. But don't capitalize the first letter of a variable name—conventionally, an initial capital letter is used to name object classes, not variables.

If you use underscores instead of capital letters to separate words in variables, as in `first_name` and `total_correct`, be consistent. Don't use `firstName` for some variables and `second_name` for others. Use one of these styles so that other programmers will find your code understandable. In some languages, variable names are case-sensitive, meaning that `firstName` and `firstname` are considered two different variables. ActionScript, however, treats them as the same thing, though it's bad form to use two different cases to refer to the same variable. If you call a variable `xPOS`, don't refer to it elsewhere as `xpos`.

Always give your variables and functions meaningful names that help you remember their purpose. Avoid meaningless names like *`foo`*, and use single-letter variables, such as *`x`* or *`i`*, only for simple things, such as the index (i.e., counting variable) in a loop. Don't confuse *`x`* and *`y`* (as used in our introductory examples, without an underscore) with *`_x`* and *`_y`*. Whereas *`x`* and *`y`* are arbitrary variable names chosen by the programmer, *`_x`* and *`_y`* are built-in properties that represent the horizontal and vertical position of, for example, a movie clip. See *MovieClip._x* in the *Language Reference* for details.

Some kinds of data in ActionScript have a recommended suffix of the form *`_suffix`* (see Table 2-1 in Chapter 2). For example, movie clips are indicated by the suffix `_mc`, text fields by `_txt`, and buttons by `_btn`. For clarity, add these to the end of your variable names, as in: `ball_mc`, or `submit_btn`.

Let's add all the labels we'll need for our quiz now, so we can use them later to walk the user through the quiz questions:

1. On the *labels* layer, select frame 1.
2. In the Property inspector, for <Frame Label>, type **`init`**.
3. At frame 10 of the *labels* layer, add a blank keyframe.
4. In the Property inspector, for <Frame Label>, type **`q2`**.
5. At frame 20 of the *labels* layer, add a blank keyframe.
6. In the Property inspector, for <Frame Label>, type **`quizEnd`**.

Scripting the Answer Buttons

Our questions are in place, our variables have been initialized, and our frames have been labeled. If we were to test our movie now, we'd see Question 1 appear with three answer buttons that do nothing when clicked, and we'd have no way for the user to get to Question 2. We need to add some code to the answer buttons so that they will advance the user through the quiz and keep track of answers along the way.

Recall that we named our button instances `choice1_btn`, `choice2_btn`, and `choice3_btn`, as shown in Figure 1-7.

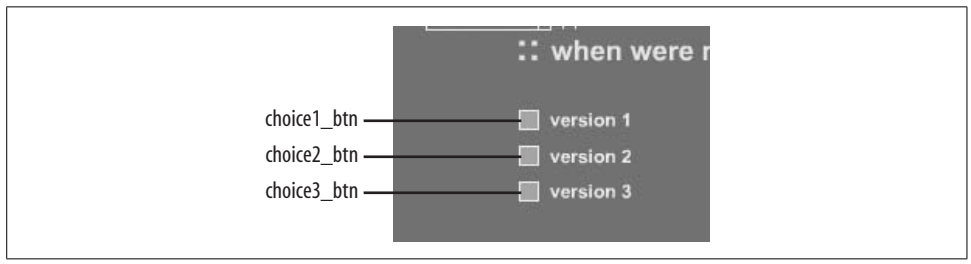


Figure 1-7. The answer buttons

To make the Question 1 buttons store the user's answer and display the next question when pressed, we'll use the code in Example 1-2.

Example 1-2. Question 1 button code

```
// Code executed when button 1 is pressed.
choice1_btn.onRelease = function ( ) {
  this._parent.q1answer = 1;
  this._parent.gotoAndStop("q2");
};

// Code executed when button 2 is pressed.
choice2_btn.onRelease = function ( ) {
  this._parent.q1answer = 2;
  this._parent.gotoAndStop("q2");
};

// Code executed when button 3 is pressed.
choice3_btn.onRelease = function ( ) {
  this._parent.q1answer = 3;
  this._parent.gotoAndStop("q2");
};
```

The code for each button consists of two statements that are executed only when a mouseclick is detected. In natural language, the code for each button says, "When the user clicks this button, make a note that he chose answer 1, 2, or 3; then proceed to Question 2." Let's dissect how it works, concentrating on the first button only:

```
choice1_btn.onRelease = function ( ) {
  this._parent.q1answer = 1;
  this._parent.gotoAndStop("q2");
};
```

Line 1 is the beginning of an *event handler*. It specifies:

- The name of the button that should respond to the event: `choice1_btn`
- The name of the event to which the button should respond: *onRelease* (which occurs when the user clicks and releases the mouse over the button)
- The function to run when the event actually happens: `function ( ) { ... }`

A function executed in response to an event is known as a *callback function*.

The opening curly brace ({) marks the beginning of the block of statements that should be executed when the *onRelease* event occurs. The end of the code block is marked by a closing curly brace (}), which is the end of the event handler's callback function.

```
choice1_btn.onRelease = function ( ) {
  ...
}
```

The event handler waits patiently for the user to click button 1. When the button is clicked, Flash executes our callback function, which contains the following two lines:

```
this._parent.q1answer = 1;
this._parent.gotoAndStop("q2");
```

The first line sets our variable `q1answer` to 1 (the other answer buttons set it to 2 or 3). The `q1answer` variable stores the user's answer for the first question. However, as a matter of good form, we provide not just the name of the `q1answer` variable, but also its location in relation to the button. The expression `this._parent` means "this button's parent movie clip" or, synonymously, "the movie clip that contains this button." The keyword `this` represents the button itself, while `_parent` is the button's parent movie clip. So, the complete statement `this._parent.q1answer = 1` means, "In this button's parent movie clip, set the variable `q1answer` to 1." If you're new to this syntax, you may find it overly verbose. However, as we'll learn in Chapter 2, every variable has a home (usually a movie clip), and we're normally required to provide not only a variable's name but also its location (e.g., `this._parent.q1answer`, as opposed to just `q1answer`).

Once we have recorded the user's answer for Question 1 in `q1answer`, we advance to Question 2 via line 2 of our callback function:

```
this._parent.gotoAndStop("q2");
```

Line 2 calls the *gotoAndStop()* function, passing it the frame label `"q2"` as an argument, which advances the movie's playhead to the frame `q2`, where Question 2 appears. However, once again we must specify, in relation to the button clicked, precisely which movie clip's playhead should move to the label `q2`. Just as with our variable, we want to control the button's parent movie clip, so we preface our *gotoAndStop()* function call with `this._parent`.

Each of our answer buttons performs the same basic tasks (sets `q1answer` and displays frame `q2`). Hence, the three button event handlers differ in only two ways:

- Each specifies a different button instance name (`choice1_btn`, `choice2_btn`, `choice3_btn`).
- Each sets a different value for `q1answer`, recording the user's actual answer to the question.

To apply the event handlers to the Question 1 buttons, enter the code from Example 1-2 into your script, below the existing *init* code on frame 1 of the *scripts* layer. When you're done, the Actions panel for frame 1 of the *scripts* layer should resemble Figure 1-8.

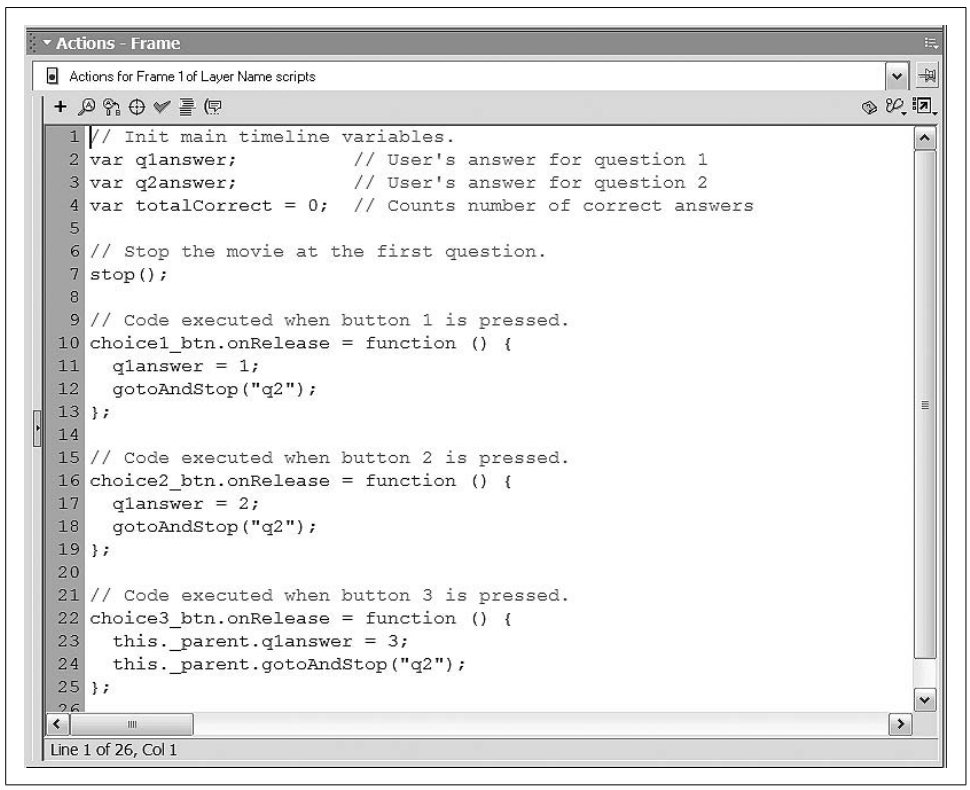


Figure 1-8. Code on frame 1

That takes care of the code for the Question 1 buttons. Let's move on to the Question 2 buttons. They work exactly like the Question 1 buttons, but they must record the user's selection for Question 2 instead of Question 1, and they must display the quiz-end screen, not the next question. Happily, we can reuse the existing buttons in our movie. We must change only the event handler functions that run when the buttons are pressed. To redefine the event handlers for the Question 2 buttons, follow these steps:

1. At frame 10 of the *scripts* layer, add a blank keyframe.
2. With frame 10 of the *scripts* layer still selected, type the code in Example 1-3 into the Actions panel.

Any script placed on a keyframe in the timeline is automatically executed when the playhead enters that frame. Hence, when Flash displays frame 10, our new button code will be applied to the Question 2 buttons.

Example 1-3. Question 2 button code

```
// Code executed when button 1 is pressed.
choice1_btn.onRelease = function ( ) {
  this._parent.q2answer = 1;
  this._parent.gotoAndStop("quizEnd");
};

// Code executed when button 2 is pressed.
choice2_btn.onRelease = function ( ) {
  this._parent.q2answer = 2;
  this._parent.gotoAndStop("quizEnd");
};

// Code executed when button 3 is pressed.
choice3_btn.onRelease = function ( ) {
  this._parent.q2answer = 3;
  this._parent.gotoAndStop("quizEnd");
};
```

Our Question 2 button event handlers are the same as they were for Question 1, except that we use the variable `q2answer` instead of `q1answer`, because we want the buttons to keep track of the user's response to Question 2. And we use `"quizEnd"` as the argument for our *gotoAndStop( )* function to advance the playhead to the end of the quiz (i.e., the frame labeled `quizEnd`) after the user answers Question 2.

Having just added event handlers to six buttons, you will no doubt have noticed how repetitive the code is. The code on each button differs from the code on the others by only a few text characters. That's not exactly efficient programming. Our button code cries out for some kind of centralized command that records the answer and advances to the next screen in the quiz. In Chapter 9, we'll see how to centralize our code with *functions*.

Building the Quiz End

Our quiz is nearly complete. We have two questions and an answer-tracking script that lets the user answer the questions and progress through the quiz. We still need a quiz-ending screen where we score the quiz and tell the user how well he fared.

To build our quiz-end screen, we need to do some basic Flash production and some scripting. Let's do the production first:

1. At frame 20 of the *question 2* layer, add a blank keyframe. This prevents Question 2 from appearing behind the contents of our quiz-end screen.
2. At frame 20 of the *choice buttons* layer, add a blank keyframe. This prevents our buttons from appearing behind the contents of our quiz-end screen.
3. At frame 20 of the *quiz end* layer, add a blank keyframe.
4. While you're still on that frame, put the following text on the Stage: "Thank you for taking the quiz." Make sure to leave some space below for the user's score.
5. At frame 20 of the *scripts* layer, add a blank keyframe.

That takes care of the production work for our quiz-end screen. Your end screen should look something like the one shown in Figure 1-9.

Now let's work on the quiz-end script. When the playhead lands on our `quizEnd` frame, we want to calculate the user's score. We need a calculation script, shown in Example 1-4, to execute when the playhead reaches frame 20. Select frame 20 of the *scripts* layer; then type the code from the example into the Actions panel.

Example 1-4. Quiz end code

```
// Tally up user's correct answers.
if (q1answer == 3) {
  totalCorrect = totalCorrect + 1;
}
if (q2answer == 2) {
  totalCorrect++;
}

// Create an onscreen text field to display the user's score.
this.createTextField("totalOutput_txt", 1, 150, 200, 200, 20);

// Show the user's score in the onscreen text field.
totalOutput_txt.text = "Your final score is: " + totalCorrect + "/2.";
```

In the calculation script, we first determine the user's score, and then we display that score on the screen. Lines 1, 9, and 12 (if you count intervening blank lines) are code comments that summarize the functionality of the sections of the script. On line 2, the first of two conditionals in our calculation script begins. In it, we put our `q1answer` variable to use. Notice that because this code is attached directly to our movie's timeline, we aren't required to supply the location of the variable `q1answer`.

```
if (q1answer == 3) {
```

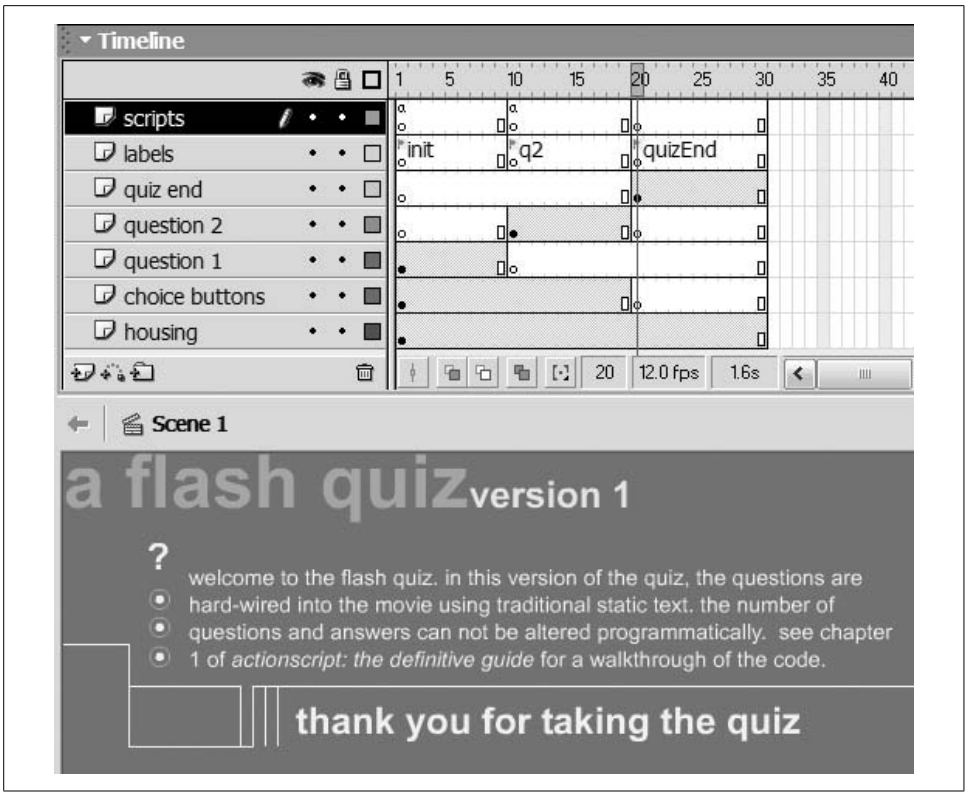


Figure 1-9. Judgment day

The keyword *if* tells the interpreter we're about to provide a list of statements that should be executed only if a certain condition is met. The terms of that condition are described in the parentheses that follow the *if* keyword: *(q1answer == 3)*, and the opening curly brace begins the block of statements to be executed conditionally. Therefore, line 2 translates into, "If the value of `q1answer` is equal to 3, then execute the statements contained in the following curly braces."

But how exactly does the condition *q1answer == 3* work? Well, let's break the phrase down. We recognize `q1answer` as the variable in which we've stored the user's answer to Question 1. The number 3 indicates the correct answer to Question 1 (movie clips first appeared in Flash version 3). The double equals sign (==) is the *equality* comparison operator, which compares two expressions. If the expression on its left (`q1answer`) equals the one on its right (3), then our condition is met and the statements within the curly braces are executed. If not, our condition is not met, and the statements within the curly braces are skipped.

Flash has no way of knowing the right answers to our quiz questions. Checking if `q1answer` is equal to 3 is our way of telling Flash to check if the user got Question 1 right. If he did, we tell Flash to add 1 to his total score as follows:

```
totalCorrect = totalCorrect + 1;
```

Line 3 says, "Make the new value of `totalCorrect` equal to the old value of `totalCorrect` plus one," (i.e., *increment* `totalCorrect`). Incrementing a variable is so common that it has its own special operator, ++.

So, instead of using this code:

```
totalCorrect = totalCorrect + 1;
```

we normally write:

```
totalCorrect++;
```

which does exactly the same thing, but more succinctly.

At line 4, a curly brace ends the block of statements to execute if our first condition is met:

```
}
```

Lines 5 through 7 are another condition:

```
if (q2answer == 2) {
  totalCorrect++;
}
```

Here we're checking whether the user answered Question 2 correctly (MP3 audio support first appeared in Flash 4). If the user chose the second answer, we add 1 to `totalCorrect` using the increment operator ++.

Because there are only two questions in our quiz, we're done tallying the user's score. For each question that the user answered correctly, we added 1 to `totalCorrect`, so `totalCorrect` contains the user's final score. The only thing left is to show the user his score in an on-screen text field. In Flash MX, we can create a text field directly with code as follows:

```
this.createTextField("totalOutput_txt", 1, 150, 200, 200, 20);
```

Here, the keyword `this` refers to the main movie timeline, which is where we want the new text field to appear. We then execute *createTextField( )*, which tells Flash to put a new text field named `totalOutput_txt` on depth 1, at an x-position of 150 and a y-position of 200, with a width of 200 and a height of 20. We can display the value of `totalCorrect` in the new text field like this:

```
totalOutput_txt.text = "Your final score is: " + totalCorrect + "/2";
```

As a new programmer, you're not expected to understand entirely how this code works. But it should give you a glimpse of the exciting stuff ActionScript can do. Try exploring on your own by changing, say, the width and height of the text field, or the text displayed in it. Text fields are covered exhaustively in the *Language Reference*, under the *TextField* class.

Testing Our Quiz

Well, that's it. Our quiz is finished. You can check whether the quiz works by using Control → Test Movie. Click on the answers in different combinations to see if your quiz is keeping score correctly. You can even create a restart button that is available throughout the quiz by making a button instance named `restart_btn` (placed on its own layer, on frame 1 of the main timeline), and adding the following code to frame 1 of the *scripts* layer:

```
restart_btn.onRelease = function ( ) {
  this._parent.totalOutput_txt.removeTextField( );
  this._parent.gotoAndStop("init");
}
```

Because `totalCorrect` is set to 0 in the code on the `init` frame, the score will reset itself each time you send the playhead to `init`.

If you find that your quiz isn't working, try comparing it with the sample quiz provided at the online Code Depot.

Onward!

So how does it feel? You've learned a bunch of phrases, some grammar, some vocabulary, and even had a drawn-out conversation with Flash (the multiple-choice quiz). Quite a rich first day of language school, I'd say.

As you can see, there's a lot to learn about ActionScript, but you can also do quite a bit with just a little knowledge. Even the amount you know now will give you plenty to play around with. Throughout the rest of this book, we'll reinforce the fundamentals you've learned by exploring them in more depth and showing them in concert with real examples. Of course, we'll also cover some topics that haven't even been introduced yet.

Remember: think communication, think cooperation, and speak clearly. And if you find yourself doing any fantastically engaging work or art that you'd like to share with others, send it over to me at *http://www.moock.org/contact/*.

Now that you have a practical frame of reference, you'll be able to appreciate and retain the foundational knowledge detailed over the next few chapters. It will give you a deeper understanding of ActionScript, enabling you to create more complex movies.

CHAPTER 2

Variables

In a typical scripted movie, we have to track and manipulate everything from frame numbers to a user's password to the velocity of a photon torpedo fired from a spaceship. In order to manage and retrieve all that information, we need to store it in *variables*, the primary information-storage containers of ActionScript.

A variable is like a bank account that, instead of holding money, holds information (*data*). Creating a new variable is like setting up a new account; we establish a place to store something we'll need in the future. And, just as every bank account has an account number, every variable has a name associated with it that is used to access the data in the variable.

Once a variable is created, we can put new data into it as often as we want—much like depositing money into an account. Or, we can find out what's in a variable by using the variable's name—much like checking an account's balance. If we no longer need our variable, we can "close the account" by deleting the variable.

The key feature to note is that variables let us refer to data that either changes or is calculated when a movie plays. Just as a bank account's number remains the same even though the account balance varies, a variable's name remains fixed even though the data it contains may change. Using that fixed reference to access changing content, we can perform complex calculations, keep track of cards in a card game, save guest book entries, or send the playhead to different locations based on changing conditions.

Is that a gleam of excitement I see in your eye? Good, I thought I might have lost you with all that talk about banks. Let's start our exploration of variables by seeing how to create them.

Creating Variables (Declaration)

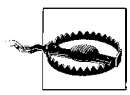
Creating a variable is called *declaration*. Declaration is the "open an account" step of our bank metaphor, where we formally bring the variable into existence. When a variable is first declared, it is empty—a blank slate waiting to be written upon. In this

state, a variable contains a special value called `undefined` (indicating the absence of data).

To declare a new variable, we use the *var* statement. For example:

```
var speed;
var bookTitle;
var x;
```

The word *var* tells the interpreter that we're declaring a variable, and the moniker that follows, such as `speed`, `bookTitle`, or `x`, becomes our new variable's name. We can create variables anywhere we can attach code: on a keyframe, a button, or a movie clip.



If you enter code in the Actions panel while on a frame that is not a keyframe, the code is attached to the nearest preceding keyframe.

We can also declare several variables with one *var* statement, like this:

```
var x, y, z;
```

However, doing so impairs our ability to add comments next to each variable.

Once a variable has been created, we can assign it a value, but before we learn how to do that, let's consider some of the subtler details of variable declaration.

Automatic Variable Creation

Many programming languages require you to declare variables before depositing data into them; failure to do so causes an error. ActionScript is not that strict. If we assign a value to a variable that does not exist, the interpreter creates a new variable for us. The bank, to continue that analogy, automatically opens an account when you try to make your first deposit.

This convenience comes at a cost, though. If we don't declare our variables ourselves, we have no central inventory to consult when examining our code. Furthermore, explicitly declaring a variable with a *var* statement can sometimes yield different results than allowing a variable to be declared *implicitly* (i.e., automatically). It's safest to declare first and use later (i.e., *explicit declaration*), as practiced throughout this book.

Legal Variable Names

Before running off to make any variables, be aware that variable names:

- Must be composed exclusively of letters, numbers, dollar signs ($) and underscores (No spaces, hyphens, or other punctuation marks are allowed.)
- Must start with a letter, an underscore (e.g., `_someVar`), or a dollar sign (e.g., `$someVar`)

- Must not exceed 255 characters (Okay, okay, that's a lie, but reevaluate your naming scheme if your variable names exceed 255 characters.)
- Are case-insensitive (Upper- and lowercase letters are treated identically, but you should be consistent nonetheless.)

These are legal variable names:

```
var first_name;
var counter;
var reallyLongVariableName;
```

These are illegal variable names that cause errors:

```
var 1first_name;                // Starts with a number
var variable name with spaces;  // Contains spaces
var another-illegal-name;       // Contains a hyphen
```

As a matter of good form, you should append suffixes to your variable names to indicate the type of information stored in the variable.

```
var firstName_str;              // _str means the variable contains a string
var products_array;             // _array means the variable contains an array
```

In Flash MX, some suffixes also activate *code hinting* in the Actions panel. For example, the suffix `_txt` in the variable name `output_txt` not only indicates that the variable stores a text field but also causes the Actions panel to display a quick-reference popup for text fields (the so-called *code hint*) when the variable name is entered. Table 2-1 lists the built-in suffixes that activate code hinting in Flash MX.

Table 2-1. Flash MX code hinting suffixes

Suffix	Datatype represented
_mc	MovieClip
_array	Array
_str	String
_btn	Button
_txt	TextField
_fmt	TextFormat
_date	Date
_sound	Sound
_xml	XML
_xmlsocket	XMLSocket
_color	Color
_video	Video
_ch	FCheckBox*
_pb	FPushButton*
_rb	FRadioButton*
_lb	FListBox*

Table 2-1. Flash MX code hinting suffixes (continued)

Suffix	Datatype represented
_sb	FScrollBar*
_cb	FComboBox*
_sp	FScrollPane*

* Flash UI Components (see Appendix G)

The programming industry's most popular alternative to Flash's suggested variable naming conventions is known as *Hungarian notation*, developed by a Hungarian Microsoft engineer, Charles Simonyi. For information, see:

http://msdn.microsoft.com/library/techart/hunganotat.htm

For a discussion of the pros and cons of Hungarian notation, see:

http://ootips.org/hungarian-notation.html

Creating dynamically named variables

Although you'll rarely, if ever, use dynamically created variable names, it's possible to generate the name of a variable programmatically. This technique was used primarily before arrays were introduced in Flash 5, so you will often encounter it in legacy code intended for Flash 4 or earlier. To create a variable name from any expression, use the *set* statement. For example, here we assign the value "bruce" to the variable `player1name`:

```
var i = 1;
set ("player" + i + "name", "bruce");
```

Arrays and objects, discussed in later chapters, provide us with a much more powerful means of tracking dynamically named data and should be used instead of dynamic variable names.

Declare Variables at the Outset

It's good practice to declare your application's variables at the beginning of every movie's main script space, which is usually the first keyframe that comes after a movie's preloader. Be sure to add a comment explaining each variable's purpose for easy identification later. The beginning of a well-organized script might look like this:

```
// ^^^^^^^^^^^^^^^^^^^^
// Initialize variables
// ^^^^^^^^^^^^^^^^^^^^
var ballSpeed;   // Velocity of ball, max 10
var score;       // Player's current score
var hiScore;     // High score (not saved between sessions)
var player1;     // Name of player 1, supplied by user
```

We can give variables an initial value at the same time we create them, as follows:

```
var ballSpeed = 5;   // Velocity of ball, default 5, max 10
var score = 0;       // Player's current score
var hiScore = 0;     // High score (not saved between sessions)
var player1 = "1P";  // Player's name defaults to 1P
```

For even tidier variable management, object-oriented programmers will want to store all variables within a class, as discussed in Chapter 12.

Assigning Values to Variables

Now comes the fun part—putting some data into our variables. If you're still playing along with the bank analogy, this is the "deposit money into our account" step. To assign a value to a variable, we use:

```
variableName = value;
```

where *variableName* is the name of a variable, and *value* is the data we're assigning to that variable. Here's an example:

```
bookTitle = "ActionScript for Flash MX: The Definitive Guide";
```

On the left side of the equals sign, the word `bookTitle` is the variable's *name* (its *identifier*). On the right side of the equals sign, the phrase "ActionScript for Flash MX: The Definitive Guide" is the variable's *value*—the datum you're depositing. The equals sign is called the *assignment* operator. It tells Flash that you want to assign (i.e., deposit) whatever is on the right of the equals sign to the variable shown on the left. If the variable on the left doesn't exist yet, Flash creates it (though relying on the interpreter to create variables implicitly isn't recommended).

Here are two more variable assignment examples:

```
speed = 25;
output = "thank you";
```

The first example assigns the integer 25 to the variable `speed`, showing that variables can contain numbers as well as text. We'll see shortly that variables can contain other kinds of data as well. The second example assigns the text "thank you" to the variable `output`. Notice that we use straight double quotation marks (" ") to delimit a text string in ActionScript.

Now let's look at a slightly more complicated example that assigns the value of the expression 1 + 5 to the variable `y`:

```
y = 1 + 5;
```

When the statement *y = 1 + 5;* is executed, 1 is first added to 5, yielding 6, and then 6 is assigned to `y`. The expression on the right side of the equals sign is *evaluated*

(calculated or resolved) before assigning the result to the variable on the left side. Here, we assign an expression that contains the variable y to another variable, z:

```
z = y + 4;
```

Once again, the expression on the right of the equals sign is evaluated, and the result is then assigned to z. The interpreter retrieves the current value of y (the interpreter checks the variable's account balance, so to speak) and adds 4 to it. Because the value of y is 6, z will be set to 10.

The syntax to assign any data—numbers, text, or any other type—to a variable is similar, regardless of the datatype. Although we haven't studied arrays yet, you should already recognize the following as a variable assignment statement:

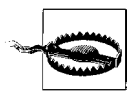
```
myList = ["John", "Joyce", "Sharon", "Rick", "Megan"];
```

As before, we put the variable name on the left, the assignment operator (the equals sign) in the middle, and the new value to assign to the variable on the right.

To assign the same value to multiple variables in a hurry, we can piggyback assignments alongside one another, like this:

```
x = y = z = 10;
```

Variable assignment always works from right to left. The preceding statement assigns 10 to z, then assigns the value of z to y, then assigns the value of y to x.



Don't confuse an equals sign, which is used to assign a value to a variable, with the algebraic equals sign you learned about in math class. Furthermore, don't confuse the equals sign (=), which is the assignment operator, with the double equals sign (==) used for comparing two expressions.

In algebra, the following makes no sense, because something can't be equal to itself plus one:

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

But in a programming language, this statement is perfectly valid and even common. It says to take the old value of x, add 1 to it, and store the new value back in the variable x. In this case, if the old value of x was 4, the statement would change x to 5.

The following statement is usually incorrect, because it changes the value of x to 5:

```
if (x = 5) { //do whatever};
```

Most likely, the programmer intended to compare the current value of x to 5:

```
if (x == 5) { //do whatever};
```

See Chapter 5 for more details on the comparison operator (==).

Changing and Retrieving Variable Values

After we create a variable, we can assign and reassign its value as often as we like, as shown in Example 2-1.

Example 2-1. Changing variable values

```
var firstName;                    // Declare the variable firstName
firstName = "Graham";             // Set the value of firstName
firstName = "Gillian";            // Change the value of firstName
firstName = "Jessica";            // Change firstName again
firstName = "James";              // Change firstName again
var x = 10;                       // Declare x and assign it a numeric value
x = "loading...please wait...";   // Assign x a text value
```

Notice that we changed the variable x's *datatype* from numeric to text data by simply assigning it a value of the desired type. Some programming languages don't allow the datatype of a variable to change, but ActionScript does.

Of course, creating variables and assigning values to them is useless if you can't retrieve the values later. To retrieve a variable's value, simply use the variable's name wherever you want its value to be used. Anytime a variable's name appears (except in a declaration or on the left side of an assignment statement), the name is converted to the variable's value.

Here are some examples. Note that _x is a built-in property representing a movie clip's horizontal position with no relation to the variable named x used in preceding examples (see the "Variable Naming Styles" sidebar in Chapter 1, and see *MovieClip._x* in the *Language Reference*).

```
newX = oldX + 5;  // Set newX to the value of oldX plus 5
ball._x = newX;   // Set the horizontal position of the
                  // ball movie clip to the value of newX
trace(firstName); // Display the value of firstName in the Output window
```

Note that in the expression ball._x, ball is a movie clip's name, and ._x indicates its x-coordinate property (i.e., horizontal position on stage). We'll learn more about properties later. The last line, *trace(firstName)*, displays a variable's value in the Output window while a script is running, which is handy for debugging your code.

Checking Whether a Variable Has a Value

Occasionally, we may wish to verify that a variable has been assigned a value before we make reference to it. As we learned earlier, a variable that has been declared but never assigned a value contains the special "nonvalue," undefined. To determine whether a variable has been assigned a value, we check if that variable's value belongs to the datatype *undefined*. For example:

```
if (typeof someVariable != "undefined") {
  // Any code placed here is executed only if someVariable is not undefined
}
```

Note the use of the *inequality operator*, `!=`, which determines whether two values are *not* equal, and the quotes around "undefined", because the *typeof* operator returns a string. In Flash MX, we can alternatively use the strict inequality operator (`!==`) to check if a value is not equal to `undefined` or not of the type *undefined*.

```
if (someVariable !== undefined) {
  // Any code placed here is executed only if someVariable is not undefined
}
```

Both the *typeof* and the strict inequality techniques just shown prevent Flash from performing automatic datatype conversion when checking if *someVariable* is `undefined`. By contrast, due to automatic datatype conversion, the regular inequality operator (`!=`) checks not only whether *someVariable* is `undefined`, but also whether it contains the `null` value. For example:

```
if (someVariable != undefined) {
  // Any code placed here is executed if someVariable is not undefined or null
}
```

The `null` value is often used by programmers to indicate that a variable is intentionally empty, but still exists.

Types of Values

The data we use in ActionScript programming comes in a variety of types. So far, we've seen numbers and text, but other types include Booleans, arrays, functions, and objects. Before we cover each datatype in detail, let's examine some datatype issues that relate specifically to variable usage.

Automatic Typing

Any ActionScript variable can contain any type of data, which may seem unremarkable, but the ability to store *any* kind of data in *any* variable is actually a bit unusual. Languages like C++ and Java use *strictly typed* variables; each variable can accept only one type of data, which must be specified when the variable is declared. ActionScript variables are *automatically* typed; when we assign data to a variable, the interpreter sets the variable's datatype for us.

Not only can ActionScript variables contain any datatype, they can also dynamically *change* datatypes. If we assign a variable a new value that has a different type than the variable's previous value, the variable is retyped automatically. So the following code is legal in ActionScript:

```
x = 1;                  // x is a number
x = "Michael";          // x is now a string
x = [4, 6, "hello"];    // x is now an array
x = 2;                  // x is a number again
```

In languages that do not perform automatic retyping, such as C++ and Java, data of the wrong type is converted to the variable's existing datatype (or it causes an error if

conversion cannot be performed). By comparison, VB.NET allows the programmer to decide whether the compiler should enforce strict typing. Strict typing may seem cumbersome, but it can prevent the errors that unsuspecting programmers may encounter due to automatic and dynamic typing, which we'll consider in the following sections.

Automatic Value Conversion

In some contexts, ActionScript expects a specific type of data. If we use a variable whose value does not match the expected type, the interpreter attempts to convert the data to the necessary type. For example, if we use a text variable where a number is needed, the interpreter will try to convert the variable's text value to a numeric value for the sake of the current operation. In Example 2-2, z is set to 2. Why? Because the subtraction operator expects a number, the value of y is converted from the string "4" to the number 4, which is subtracted from 6 (the value of x), yielding the result 2.

Example 2-2. Automatic string-to-number conversion

```
x = 6;      // x is a number, 6
y = "4";    // y is a string, "4"
z = x - y;  // This sets z to the number 2
```

Conversely, if we use a numeric variable where a string is expected, the interpreter attempts to convert the number to a string. In Example 2-3, z is set to the string "64", not the number 10. Why? Because the second operand in the expression x + y is a string. Therefore, the + operator performs string concatenation instead of mathematical addition. The value of x (6) is converted to the string "6" and then concatenated with the string "4" (the value of y), yielding the result "64".

Example 2-3. Automatic number-to-string conversion

```
x = 6;      // x is a number, 6
y = "4";    // y is a string, "4"
z = x + y;  // This sets z to the string "64"
```

The automatic type conversion that occurs when evaluating a variable as part of an expression is performed on a *copy* of the variable's data—it does not affect the original variable's type. A variable's type changes only when the variable is assigned a data value that does not match its previous value's type. So, at the conclusion of Example 2-2 and Example 2-3, y remains a string, and x remains a number.

Notice that the operator on line 3 (- in Example 2-2, + in Example 2-3), has a profound impact on the value assigned to z. In Example 2-2 the string "4" becomes the number 4, whereas in Example 2-3 the opposite occurs (the number 6 becomes the string "6"), because the rules for datatype conversion are different for the + operator than for the - operator. We'll cover data conversion rules in Chapter 3 and operators in Chapter 5.

Determining the Type Manually

Automatic datatyping and conversion can be convenient, but, as Example 2-2 and Example 2-3 illustrate, they may also produce unexpected results. Before performing commands that operate on mixed datatypes, you may wish to determine a variable's datatype using the *typeof* operator:

```
productName = "Macromedia Flash";  // String value
trace(typeof productName);         // Displays: "string"
```

Once we know a variable's type, we can proceed conditionally. Here, for example, we check whether a variable is a number before proceeding:

```
if (typeof age == "number") {
  // okay to carry on
} else {
  trace ("Age isn't a number");  // Display an error message
}
```

For full details on the *typeof* operator, see Chapter 5.

Variable Scope

Earlier we learned how to create variables and retrieve their values. But all our variables were attached to a single frame of the main timeline of a Flash document. When a document contains multiple frames and multiple movie clip timelines, variable creation and value retrieval becomes a little more complicated.

Timeline Variables

To illustrate the use of timeline variables, let's consider two simple scenarios.

Scenario 1: Accessing a value defined earlier on the same timeline

What happens when we define a variable in one frame of a timeline and try to access it later?

Suppose we create a variable, `x`, in frame 1 of the main timeline. After creating `x`, we set its value to 10:

```
var x;
x = 10;
```

Then, in the next frame (frame 2), we attach the following code:

```
trace(x);
```

When we play our movie, does anything appear in the Output window? We created our variable in frame 1, but we're attempting to retrieve its value in frame 2; does our variable still exist? Yes. The Output window displays 10.

When you define a variable on a timeline, that variable is accessible from all the other frames of that timeline.

Scenario 2: Accessing a value defined later on the same timeline

What happens if we try instead to access a variable before the frame in which it is assigned a value?

Suppose we add the following code to frame 30 of a movie's main timeline:

```
password = "let_me_in";
gotoAndStop(15);
```

and on frame 15 we add:

```
trace(password);
```

When the movie plays, we see the following in the Output window:

```
undefined
let_me_in
```

Here's why: when the movie's playhead reaches frame 15, Flash displays the value of `password` in the Output window. The first time through, `password` does not exist, so Flash displays `undefined`. However, when the playhead reaches frame 30, `password` is created and assigned the value "let_me_in". Then the playhead is sent back to frame 15 (*gotoAndStop(15)*), causing frame 15's code to execute again. This time, even though *password* is defined on a later frame than our *trace(password)* statement, it is still part of the same timeline; so, its value exists, and Flash displays it in the Output window.

Any variable declared on a timeline is available to all the scripts of its timeline for as long as that timeline exists. However, a variable's value is not defined until the script that assigns it a value is reached. Once the value of a variable is set, the value is maintained, even if the playhead jumps backward in the timeline. In other words, the value of a variable is determined by the order in which scripts are executed, as determined by the movement of the playhead, not the order of the frames to which scripts are attached on the timeline.

Variable Accessibility (Scope)

The two scenarios we just presented explore issues of *scope*. A variable's scope describes when and where the variable can be manipulated by the code in a movie. Scope defines a variable's life span and its accessibility to other blocks of code in our scripts. To determine a variable's scope, we must answer two questions: (a) how long does the variable exist? and (b) from where in our code can we set or retrieve the variable's value?

In traditional programming, variables are often broken into two general scope categories: *global* and *local*. Variables that are accessible throughout an entire program are called *global variables*. Variables that are accessible only to limited sections of a program are called *local variables*. In addition to these two conventional variable types, Flash adds *timeline variables* that are scoped to individual movie clip instances. Flash 5 supported conventional local variables but did not support true global variables; Flash MX introduces support for true global variables.

In Flash, all variables are scoped to one of the following:

- A function (local variable)
- The main timeline or a movie clip (timeline variable)
- In Flash MX, the `_global` object (global variable)

Movie Clip Variables and Global Variables

As shown in the two earlier scenarios, a variable defined on a timeline is available to all the scripts on that timeline—from the first frame to the last frame. But what happens if we have more than one timeline in a movie, as described in Scenario 3?

Scenario 3: Variables declared on separate timelines

Suppose we have two basic geometric shapes, a *square* and a *circle*, defined as movie clip symbols. Recall that each movie clip maintains its own independent timeline.

On frame 1 of the *square* clip symbol, we set the variable `x` to 3:

```
var x;
x = 3;
```

On frame 1 of the *circle* clip symbol, we set the variable `y` to 4:

```
var y;
y = 4;
```

We place instances of those clips on frame 1, layer 1 of the main timeline, and we name our instances `square` and `circle`.

First question: if we attach the following code:

```
trace(x);
trace(y);
```

to frame 1 of the *main* movie timeline (upon which `square` and `circle` have been placed), what appears in the Output window when the movie plays?

Answer:

```
undefined
undefined
```

The variable x is defined on the square timeline, and y is defined on the circle timeline; neither are defined on our main timeline. Because Flash cannot find any variable named x or y on the main timeline, it displays undefined in the Output window.

Variables attached to a movie clip timeline (like that of square or circle) have scope limited to that timeline. They are not directly accessible to scripts on other timelines, such as our main movie timeline.

Second question: if we place the *trace(x)* and *trace(y)* statements on frame 1 of our square movie clip instead of frame 1 of our main movie timeline, what will appear in the Output window?

Answer: the value of x, which is 3, and the value of y, undefined.

```
3
undefined
```

The value of x is displayed as 3, because x is defined on the timeline of square and is therefore accessible to the *trace()* command, which also resides on that timeline. But the value of y doesn't appear in the Output window, because y is defined in circle, which is a separate timeline.

In Flash, a variable attached to an individual timeline (i.e., any movie clip) is directly accessible to the scripts on that timeline only. Furthermore, two or more movie clips can legitimately define a variable with the same name. For example, we could create a variable named x in circle with a value of 25, even though there already is a variable named x in square with a value of 3. The result of *trace(x)* from square would be 3, but from circle it would be 25.

We refer to variables attached to timelines as *timeline variables* or *movie clip variables*. Now let's create a *global variable* that is directly accessible to all the scripts in a movie (any timeline). We'll put the following code on frame 1 of the square timeline, even though the code would have the same effect from any movie clip:

```
_global.day = "Monday";
```

Now we place this code on frame 1 of the main movie:

```
trace(day);
```

Because the day variable is global, Flash can find it from the main timeline, even though it was created in square, and the Output window displays:

```
Monday
```

As we'll see in "The Scope Chain," later in this chapter, when a timeline variable and a global variable both have the same name, the timeline variable takes precedence. For example, if we create a timeline variable, day, inside our circle movie clip, as follows:

```
var day = "Friday";
```

and then we check the value of day inside circle:

```
trace(day);
```

the Output window displays "Friday" (the timeline variable's value), not "Monday" (the global variable's value). For more information on global variables, see _global in the *Language Reference*.

Though true global variables were not supported in Flash 5, it was possible to simulate them using the *Object* class or *MovieClip* class. To create a variable that is available on all timelines in Flash 5, use the following statement:

```
MovieClip.prototype.myGlobalVariable = myValue;
```

For example:

```
MovieClip.prototype.msg = "Hello world";
```

Alternatively, assign the variable as a property of *Object*, as follows:

```
Object.msg = "Hello world";
```

Then, from any other timeline, access msg like this:

```
trace(Object.msg);
```

We discuss this technique (and the reason it works) in "The end of the inheritance chain," in Chapter 12. In Flash MX, the _global object is the preferred way to store global variables.

Accessing Variables on Different Timelines

Even though variables on one timeline are not accessible directly to the scripts on other timelines, they are accessible indirectly. To create, retrieve, or assign a variable on a separate timeline, we use *dot syntax*, a standard notation common to object-oriented programming languages such as Java, C++, and JavaScript. Here's the generic dot syntax phrasing we use to address a variable on a separate timeline:

```
movieClipInstanceName.variableName
```

That is, we refer to a variable on another timeline using the name of the movie clip that contains the variable, followed by a dot, and finally the variable name itself. In our earlier scenario, for example, from the main timeline we can refer to the variable x in the square clip as:

```
square.x
```

Also from the main timeline, we can refer to the variable y in the circle clip as:

```
circle.y
```

A reference to a variable that includes the variable's location is known as a *qualified reference*. Qualified references tell Flash where to find the variable we're referring to; unqualified references require Flash to make assumptions about where the variable resides, as discussed later in "The Scope Chain." Hence, it's always good form to use qualified references to variables.

We can use qualified references from our main movie timeline to assign and retrieve variables in square like this:

```
square.z = 5;       // Assign 5 to z in square
var mainZ;          // Create mainZ on the main timeline
mainZ = square.z;   // Assign the value of z in square to mainZ
```

However, using the *movieClipInstanceName.variableName* syntax, we can't refer to variables in square from our circle clip. If we put a reference to square.x on a frame in circle, the interpreter tries to find a clip called square *inside* of circle, but square lives on the main timeline. So, we need a mechanism that lets us refer to the timeline that contains the square clip (in this case, the main timeline) from the circle clip. That mechanism comes in the form of two special properties: _root and _parent.

The _root and _ parent properties

The _root property is a direct reference to the main timeline of a movie. From any depth of nesting in a movie clip structure, we can always address variables on the main movie timeline using _root, like this:

```
_root.mainZ       // Access the variable mainZ on the main timeline
_root.firstName   // Access the variable firstName on the main timeline
```

We can even combine a reference to _root with references to movie clip instances, drilling down the nested structure of a movie in the process. For example, we can reference the variable x, inside the clip square that resides on the main movie timeline, as:

```
_root.square.x
```

This reference works from anywhere in our movie, no matter what the depth of clip nesting is, because the reference starts at our main movie timeline, _root. Here's another nested example showing how to access the variable area in the instance triangle that resides on the timeline of the instance shapes:

```
_root.shapes.triangle.area
```

Any reference to a variable that starts with the _root keyword is called an *absolute reference* or a *fully qualified reference*, because it describes the location of the variable in relation to a fixed, immutable point in our movie: the main timeline.

There are times, however, when we want to refer to variables on other timelines without referring to the main timeline of a movie. To do so, we use the _parent property, which refers to the timeline upon which the current movie clip instance resides. For example, from code attached to a frame of the clip square, we can refer to variables on the timeline that contains square using this syntax:

```
_parent.myVariable
```

References that start with the keyword _parent are called *relative references*, because they are resolved relative to the location of the clip in which they occur.

Suppose we have a variable, size, defined on the main timeline of a movie. We place a clip instance named shapes on our main movie timeline, and on the shapes timeline we define the variable color. We also place a clip named triangle on the shapes timeline, as shown in Figure 2-1.

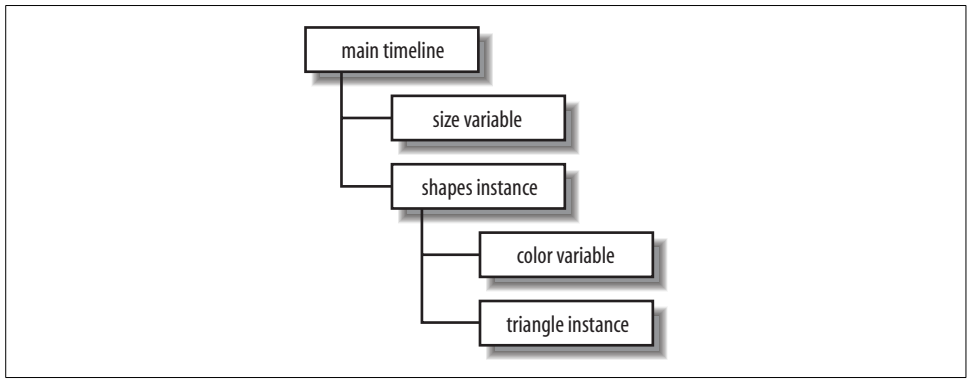


Figure 2-1. A sample movie clip hierarchy

To display the value of the variable color (which is in the shapes clip) from code attached to the timeline of triangle, we could use an absolute reference starting at the main timeline, like this:

```
trace(_root.shapes.color);
```

But that ties our code to the main movie timeline. To make our code more flexible, we should instead use the _parent property to create a relative reference, like this:

```
trace(_parent.color);
```

Our first approach (using _root) works from the top down; it starts at the main timeline and descends through the movie clip hierarchy until it reaches the color variable. The second approach (using _parent) works from the bottom up; it starts with the clip that contains the *trace()* statement (the triangle clip), and ascends one level up the clip structure, which happens to be the shapes clip, where it finds the color variable.

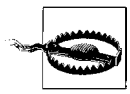
We can use _parent twice in a row, separated by a dot, to ascend the hierarchy of clips and access our size variable on the main timeline. Here we attach some code to triangle that refers to size on the main movie timeline:

```
trace(_parent._parent.size);
```

Using the _parent property twice in succession ascends two levels up the movie clip hierarchy, which in this context brings us to the main timeline of the movie.

Your approach to variable addressing will depend on what you want to happen when you place instances of a movie clip symbol on various timelines. In our triangle example, if we wanted our reference to color to always point to color as defined in

the shapes clip, we would use the _root syntax, which gives us a fixed reference to color in shapes. But if we wanted our reference to color to refer to a different color variable, depending on which timeline held a given triangle instance, we would use the _parent syntax.



The meaning of _root changes when a *.swf* file is loaded into a movie clip. For example, suppose we load *myPictures.swf* into a movie clip in *container.swf*. When *myPictures.swf* runs independently, _root means the main timeline of the *myPictures.swf* file. But when *myPictures.swf* runs inside a movie clip of *container.swf*, _root means the main timeline of *container.swf*.

Flash MX makes this problem easy to solve using a global variable. To retain a consistent reference to the main timeline of a movie, even when the movie is loaded into a movie clip, place the following code on the main timeline:

```
_global.movieNameMainTimeline = this;
```

For example:

```
_global.myPicturesMainTimeline = this;
```

This command creates a global reference to the main timeline, which allows you to refer to variables on the main timeline from any other timeline using:

```
myPicturesMainTimeline.variableName
```

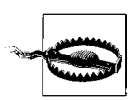
instead of:

```
_root.variableName
```

Accessing variables on different document levels

The _root property refers to the main movie timeline of the current level (i.e., the current document), but the Flash Player can accommodate multiple documents in its *document stack*. The main timeline of any movie loaded in the Player document stack can be referenced using _level*n*, where *n* is the number of the level on which the movie resides. Level numbering begins with 0, such as _level0, _level1, _level2, _level3, and so on. For information on loading multiple movies, see Chapter 13. Here are some examples of multiple-level variable references:

```
_level1.firstName         // firstName on _level1's main timeline
_level4.ball.area         // area in ball clip on _level4's main timeline
_level0.guestBook.email   // email in guestBook clip on _level0's main timeline
```



When referring to variables across movie clip instances, make sure that you have named your clip instances using the <Instance Name> field in the Property inspector and referred to them by the same name in your code. Don't confuse the symbol name in the Library with the instance name on the Stage. If your instances are not named, your code cannot refer to them by name. Unnamed instances and misspelled instance names are extremely common sources of problems.

Prior to introducing support for dot syntax in Flash 5, Flash supported an older "path" style syntax that used backslashes instead of dots. Flash MX still supports the legacy syntax, but you should update it to the more modern dot syntax as described in Table C-2 in Appendix C.

Movie Clip Variable Life Span

Earlier, we said that the scope of a variable answers two questions: (a) how long does the variable exist? and (b) from where in our code can we set or retrieve the variable's value? For movie clip variables, we have shown the factors involved in answering the second question. But we skipped answering the first question. Let's return to it now with one final variable-coding scenario.

Scenario 4: Life span of movie clip variables

Suppose we create a new movie with keyframes at frames 1, 2, and 3. On frame 1, we place a clip instance, `ball`. On the `ball` timeline, we create a variable, `radius`. Assume frame 3 of our main timeline is a blank keyframe (the `ball` instance is not present there).

From frame 2 of the main movie timeline, we can find out the value of `radius` using this code:

```
trace(ball.radius);
```

Here's the question: if we move this line of code from frame 2 to frame 3 of the main timeline, what appears in the Output window when our movie plays?

Answer: `undefined`. When the `ball` clip is removed from the main timeline at frame 3, all its variables are destroyed in the process.

Movie clip variables last only while the clip in which they reside is present on stage. Variables defined on the main timeline of a Flash document persist within each document but are lost if the document is unloaded from the Player (either via the *unloadMovie()* function or because another movie is loaded into the movie's level).

A variable's life span is important when scripting movies that contain movie clips placed across multiple frames on various timelines. Always make sure that any clip you're addressing is present on a timeline before you try to use the variables in that clip.

Local Variables

Movie clip variables are scoped to movie clips and persist as long as the movie clip on which they are defined exists. Sometimes, that's longer than we need them to live. For situations in which we need a variable only temporarily, ActionScript offers

variables with *local* scope (i.e., *local variables*), which live for a much shorter time than normal movie clip variables.

Local variables are used in functions. If you haven't worked with functions before, you should skip the rest of this section and come back to it once you've read Chapter 9.

Functions often employ variables that are needed only within the function. For example, suppose we have a function that displays the elements of an array:

```
function displayElements(theArray) {
   var counter = 0;
   while(counter < theArray.length) {
      trace("Element " + counter + ": " + theArray[counter]);
      counter++;
   }
}
```

The `counter` variable is required to display the elements of the array, but it has no use thereafter. We could leave it defined on the timeline, but that's bad form for two reasons: (a) if `counter` persists, it takes up memory during the rest of our movie, and (b) if `counter` is accessible outside our function, it may conflict with other variables named `counter`. We would, therefore, like `counter` to die after the *displayElements()* function has finished.

To cause `counter` to be deleted automatically at the end of our function, we define it as a *local variable*. Unlike movie clip variables, local variables are automatically marked for *deallocation* (removal from memory) by the interpreter when the function that defines them finishes.

To specify that a variable should be local, declare it with the *var* keyword from inside your function, as in the preceding *displayElements()* example.

Take heed, though; when placed *outside* of a function, the *var* statement creates a normal timeline variable, not a local variable. As shown in Example 2-4, the location of the *var* statement makes all the difference.

Variables within functions need not be local. We can create or change a movie clip variable from inside a function by omitting the *var* keyword. If we do not use the *var* keyword, but instead simply assign a value to a variable from within a function, Flash treats that variable as a nonlocal variable under some conditions. Consider this variable assignment inside a function:

```
function setHeight () {
   height = 10;
}
```

The effect of the statement `height = 10;` depends on whether `height` is a local variable or movie clip variable. If `height` is a previously declared local variable (which it is not in the example at hand), the statement `height = 10;` simply modifies the local variable's value. If there is no local variable named `height`, as in this case, the inter-

preter creates a movie clip (nonlocal) variable named `height` and sets its value to 10. As a nonlocal variable, `height` persists even after the function finishes.

Example 2-4 demonstrates local and nonlocal variable usage.

Example 2-4. Local and nonlocal variables

```
var x = 5;                          // New nonlocal variable, x, is now 5
function variableDemo () {
   x = 10;                          // Nonlocal variable, x, is now 10
   y = 20;                          // New nonlocal variable, y, is now 20
   var z = 30;                      // New local variable, z, is now 30
   trace(x + "," + y + "," + z);    // Display values in Output window
}
variableDemo();  // Call our function. Displays: 10,20,30
// Now check the values of x, y, and z after the function has finished.
trace(x);         // Displays: 10 (reassignment in our function was permanent)
trace(y);         // Displays: 20 (nonlocal variable, y, still exists)
trace(z);         // Displays "undefined" in Flash MX or nothing in Flash 5
                  // (former local variable, z, is undefined)
```

Note that it is possible (though confusing and ill-advised) to have both a local and a nonlocal variable that share the same name within a script but have different scopes. Example 2-5 shows such a case.

Example 2-5. Local and nonlocal variables with the same name

```
var myColor = "blue";
function hexRed () {
   var myColor = "#FF0000";
   return myColor;
}
trace(hexRed());  // Displays: #FF0000 (the local variable myColor)
trace(myColor);    // Displays: "blue" (setting the local variable,
                   // myColor, to #FF0000 did not affect the nonlocal version)
```

The Scope Chain

Internally, Flash resolves each unqualified variable reference in our code using a *scope chain*, which is simply a hierarchy of places (scopes) to look when a reference to a variable is *resolved* (i.e., when the correct variable is located unambiguously and its value is retrieved). Whenever Flash encounters a variable in a script or function, it seeks the variable's value in the scope chain associated with that code. The hunt for the variable ends when the variable is found somewhere in the scope chain or when it is not found at all (the variable is `undefined`.) The scope chain for a frame script (code attached directly to a movie clip keyframe) includes two scopes, which are traditionally read from the bottom up:

- Global object (interpreter looks here last)
- Enclosing movie clip object (interpreter looks here first)

When a variable is referenced in a frame script, Flash checks the current movie clip for the variable. If the variable is not found in the movie clip (or its prototype), Flash checks for a global variable of that name. If no global variable is found, Flash returns `undefined`. When both a movie clip variable and a global variable of the same name are defined, Flash retrieves the movie clip variable, because it is first in the search order for the scope chain. Hence, a movie clip variable always overrides a global variable of the same name.

By contrast, the scope chain for code in a nonnested function includes three scopes (again, read from the bottom up):

- Global object (`_global`)
- Enclosing movie clip object
- Local variables

When a variable is referenced from within a function, Flash checks first for a local variable—defined with the *var* statement—within the function, as shown in Example 2-5. If no local variable is found, Flash checks for a movie clip variable in the movie clip in which the function was defined. (We'll consider this topic, including the behavior for nested functions, more thoroughly in Chapter 9.) If the variable is not found in that movie clip (or its prototype), Flash checks for a global variable of the same name. If no global variable is found, Flash returns `undefined`.

Using the *with* statement, we can add any object to the scope chain as the first place Flash should look for variables. For details, see Chapter 5.

Until you're familiar with the scope chain, you may want to qualify your variable name references (e.g., `myGameMainTimeline.numPlayers` versus `numPlayers`) to tell Flash explicitly where to look for a variable. Qualified references are unambiguous and are therefore easiest to understand for all levels of programmers. See "Accessing Variables on Different Timelines," earlier in this chapter.

Don't confuse the scope chain with the *prototype chain*. Flash uses the scope chain to look up variables; it uses the prototype chain to look up properties and methods. We'll discuss the prototype chain in Chapter 12.

Note that, technically, each scope chain is implemented internally as a series of objects used by the interpreter to look up variables. See Chapter 9 of this book and David Flanagan's eloquent description of JavaScript's variable scope in *JavaScript: The Definitive Guide* (O'Reilly & Associates, Inc.). Bear in mind, however, that ActionScript adds movie clips to JavaScript's global and local execution contexts.

Event Handler Scope

As you'll see in Chapter 10, different kinds of event handlers have different scopes in Flash. Table 2-2 provides a summary of event handler scope. For each type of handler, an unqualified, nonlocal variable declaration (one without a *var* statement) will

create a new variable in the movie clip listed in the *Scope chain* column of the table. We'll examine this topic more closely in Chapter 10.

Table 2-2. Event handler scope

Event handler type	Scope chain
Callback function	Global object
	Clip in which the function was defined
	Local variables
Listener method	Global object
	Clip in which the function was defined
	Local variables
Movie clip with *on( )* or *onClipEvent( )* handler	Global object
	Clip that physically bears the handler at authoring time
Button with *on( )* handler	Global object
	Clip on whose timeline the button resides

Loading External Variables

While most variables are created directly inside Flash, it's also common to load variables from an external text file, server script, or web page. Loading external variables does not mean simply assigning a value from an external source to existing variables. Loading external variables actually creates new variables at runtime. The following entries in the *Language Reference* explain various variable-loading techniques:

- *LoadVars*
- *loadVariables( )*
- *fscommand( )*

Additionally, in some browsers, JavaScript can set a variable in Flash at runtime using the syntax:

```
movieObj.SetVariable("/someClip:firstName", "Colin");
```

where *movieObj* is an object reference to the Flash movie embedded in the page. This technique works only in the following browsers:

- Internet Explorer on Windows
- Netscape 4 on Windows, Macintosh, and Linux
- Netscape 6.2 with Flash Player 6.0.40.0 (or higher) on Windows, Macintosh, and Linux

For complete details, see:

http://www.moock.org/webdesign/flash/fscommand/

Similarly, an HTML document's <OBJECT> and <EMBED> tags can create variables in a Flash movie, when the movie initially loads, via the FlashVars parameter or a query string. Variables passed to a Flash movie via FlashVars or a query string must be URL-encoded according to the rules described in the *LoadVars* class in the *Language Reference*. The following code uses the FlashVars parameter to create the variables authorName and bookName with the values "Colin Moock" and "ASDG" on the main timeline of a movie called *main.swf*:

```
<OBJECT classid="clsid:D27CDB6E-AE6D-11cf-96B8-444553540000"
 CODEBASE="http://download.macromedia.com/pub/shockwave/cabs/flash/swflash.cab"
 WIDTH="550"
 HEIGHT="400">
 <PARAM NAME=movie VALUE="main.swf">
 <PARAM NAME=FlashVars VALUE="authorName=Colin+Moock&bookName=ASDG">

 <EMBED src="main.swf"
 FlashVars="authorName=Colin+Moock&bookName=ASDG"
 WIDTH="550"
 HEIGHT="400"
 TYPE="application/x-shockwave-flash"
 PLUGINSPAGE="http://www.macromedia.com/go/getflashplayer">
 </EMBED>
</OBJECT>
```

The FlashVars parameter requires Flash Player 6 or later and can pass a maximum of 63 KB of data to Flash. To pass data from HTML to earlier Flash Players, append the parameters to the URL of the *.swf* filename, as shown in the following example. In this case, the data transfer limit depends on the web server.

```
<OBJECT classid="clsid:D27CDB6E-AE6D-11cf-96B8-444553540000"
 codebase="http://download.macromedia.com/pub/shockwave/cabs/flash/swflash.cab"
 WIDTH="550"
 HEIGHT="400">
 <PARAM NAME=movie VALUE="main.swf?authorName=Colin+Moock">

 <EMBED src="main.swf?authorName=Colin+Moock"
 WIDTH="550"
 HEIGHT="400"
 TYPE="application/x-shockwave-flash"
 PLUGINSPAGE="http://www.macromedia.com/go/getflashplayer">
 </EMBED>
</OBJECT>
```

The passed variables are always created on the movie's main timeline. There is no direct way to set variables in a nested movie clip using FlashVars or a query string. See Appendix H for more details on the <OBJECT> and <EMBED> tags.

Some Applied Examples

We've had an awful lot of variable theory. The following examples provide three variable-centric code samples that show some of these concepts in use. Refer to the comments within the code for an explanation of each line of code.

Example 2-6 sends the playhead of a movie clip to a random destination.

Example 2-6. Send the playhead to a random frame on the current timeline

```
var randomFrame;                // Stores the randomly picked frame number
var numFrames;                  // Stores the total number of frames in the clip
numFrames = this._totalframes;  // Store the current movie clip's
                                // _totalframes property in numFrames
// Pick a random frame
randomFrame = Math.floor(Math.random( ) * numFrames + 1);
this.gotoAndStop(randomFrame);  // Send playhead of current movie clip to
                                // chosen random frame
```

Example 2-7 determines the distance between two clips. A working version of this example is available from the online Code Depot.

Example 2-7. Calculate the distance between two movie clips

```
var c;                  // A convenient reference to the circle clip object
var s;                  // A convenient reference to the square clip object
var deltaX;             // The horizontal distance between c and s
var deltaY;             // The vertical distance between c and s
var dist;               // The total distance between c and s
c = _root.circle;       // Get reference to the circle clip
s = _root.square;       // Get reference to the square clip
deltaX = c._x - s._x;   // Compute the horizontal distance between the clips
deltaY = c._y - s._y;   // Compute the vertical distance between the clips
// The distance is the square root of (deltaX squared plus deltaY squared).
dist = Math.sqrt((deltaX * deltaX) + (deltaY * deltaY));
// Using variables, the code above creates tidy references that
// are much more readable than the alternative, shown below:
dist = Math.sqrt(((_root.circle._x - _root.square._x) * (_root.circle._x -
_root.square._x)) + ((_root.circle._y - _root.square._y) * (_root.circle._y -
_root.square._y)));
```

Example 2-8 converts from Fahrenheit to Celsius. A working version showing bi-directional conversion is available in the online Code Depot.

Example 2-8. A Fahrenheit/Celsius temperature converter

```
// Create variables
var fahrenheit;         // Temperature in Fahrenheit
var result;             // The value resulting from conversion

// Initialize variables
fahrenheit = 451;       // Set a Fahrenheit temperature

// Calculate the Celsius equivalent
result = (fahrenheit - 32) / 1.8;
// Display the result
trace (fahrenheit + " degrees Fahrenheit is " + result + " degrees Celsius.");
```

Onward!

Now that we know the fundamentals of storing information in variables, including advanced topics such as variable scope, it's time we learn something more about the content that variables store: *data*. Over the next three chapters, we'll learn what data is, how it can be manipulated, and why it's an essential part of nearly everything we build with ActionScript.

CHAPTER 3

Data and Datatypes

Having worked with variable values in Chapter 2, you've already had a casual introduction to data, the information we manipulate in our scripts. In this chapter, we'll explore data in more depth, learning how ActionScript defines, categorizes, and stores data. We'll also explore how to create and classify data.

Data Versus Information

In the broadest sense, *data* is anything that can be stored by a computer, from words and numbers to images, video, and sound. All computer data is stored as a sequence of 1s and 0s, which you might recognize from high-tech marketing materials:

```
01010101010101011010101101101010101010100000101010101011010101010
10101010101010101110101010101010101010101010101111101010101010101
01010101010101010101010101110101010101010101010101010101010101010
```

Data is information in its crude state—raw and meaningless. Semantics give information meaning. Consider, for example, the number 8008898969. As raw data it isn't very meaningful, but when we classify it semantically as the telephone number (800) 889-8969, the data becomes useful information.

This chapter shows how to add meaning to raw computer data so that it becomes human-comprehensible information.

Retaining Meaning with Datatypes

How do we store information as raw data without losing meaning? By categorizing our data and defining its datatype, we give it context that defines its meaning.

For example, suppose we have three numbers: 5155534, 5159592, and 4593030. By categorizing our data—as, say, a phone number, fax number, and parcel tracking number—the context (and, hence, the meaning) of our data is preserved. When categorized, each of the otherwise-nondescript seven-digit numbers becomes meaningful.

Programming languages use *datatypes* to provide rudimentary categories for data. For example, nearly all programming languages define datatypes to store and manipulate text (a.k.a. *strings*) and numbers. To distinguish between multiple numbers, we can use well-conceived variable names, such as `phoneNumber` and `faxNumber`. In more complex situations, we can create our own custom data categories with *objects* and *object classes*, as covered in Chapter 12. Before we think about making our own data categories, let's see which categories come built into ActionScript.

The ActionScript Datatypes

When programming, we may want to store a product name, a background color, or the number of stars to be placed in a night sky. We use the following ActionScript datatypes to store our data:

string
: For text sequences such as "`hi there`". A *string* is a series of characters (alphanumerics and punctuation).

number
: For numbers, such as 351 and 7.5. Numbers are used for counting and for mathematical equations.

boolean
: For logical decisions. With Boolean data, we can represent or record the status of some condition or the result of some comparison. Boolean data has only two legal values: `true` and `false`.

null and undefined
: For representing an *absence* of data, ActionScript provides two special data values: `null` and `undefined`. You can think of them as the only permissible values of the *null* and *undefined* datatypes.

array
: For lists of one or more pieces of data.

movieclip
: For manipulating movie clip instances.

object
: For arbitrary built-in or user-defined classes of data.

Every piece of data we store in ActionScript will fall into one of these categories. Before studying each datatype in Chapter 4, we'll consider the general issues that affect our use of all data.

Creating and Categorizing Data

There are two ways to create a new datum with ActionScript, and both methods require the use of *expressions*—phrases of code that represent data in scripts.

A literal expression (or *literal* for short) is a series of letters, numbers, and punctuation that *is* the datum. A data literal is a verbatim description of data in a program's source code. This contrasts with a *variable*, which is merely a container that holds a datum. Each datatype defines its own rules for the creation of literals. For example, string literals are enclosed in quotes, whereas numeric literals are not. Here are some examples of literals:

```
"loading...please wait"  // A string literal
1.51                     // A numeric literal
["jane", "jonathan"]     // An array literal
{x: 10, y: 15}           // An object literal
```

Note that movie clips cannot be represented by literals but are referred to by instance names.

We can also generate data programmatically with a *complex expression*. Complex expressions are phrases of code with a value that must be calculated or computed, not taken literally. The calculated value is the datum being represented. For example, each of these complex expressions results in a single datum:

```
1999 + 1       // Yields the numeric datum 2000
"1999" + "1"   // Yields the string datum "19991"
"hi " + "ma!"  // Yields the string datum "hi ma!"
firstName      // Yields the value of the variable firstName
_currentframe  // Yields the frame number of the playhead's current position
new Date()     // Yields a new Date object with the current date and time
```

Notice that an individual literal expression, such as 1999 or 1, can be a valid part of a larger complex expression, as in *1999 + 1*.

Whether we use a literal expression or a complex expression to create data, we must store every datum that we want to use later. The result of the expression `"hi" + "ma!"` is lost unless we store it, say, in a variable. For example:

```
// This datum is fleeting and dies immediately after it's created
"hi " + "ma";
// This datum is stored in a variable and can be
// accessed later via the variable welcomeMessage
var welcomeMessage = "hi " + "ma!";
```

How do we categorize data into the appropriate type? That is, how do we specify that a datum is a number, a string, an array, or whatever? In most cases, we don't categorize new data ourselves; the ActionScript interpreter automatically assigns or infers each datum's type based on a set of internal rules.

Automatic Literal Typing

The interpreter infers a literal datum's type by examining its syntax, as explained in the comments in the following code fragment:

```
"animal"             // Quotation marks identify "animal" as a string
1.35                 // If it contains only integers and a decimal point,
                     // it is a number
```

```
true                // Special keyword true identifies this as a Boolean
null                // Special keyword null identifies this as the null type
undefined           // Special keyword undefined identifies the undefined type
["hello", 2, true]  // Square brackets and values separated by commas
                    // indicate that this is an array
{x: 234, y: 456}    // Curly braces and property name/value pairs separated
                    // by commas indicate that this is an object
```

As you can see, using correct syntax with data literals is extremely important. Incorrect syntax may cause an error or result in the misinterpretation of a datum's content. For example:

```
animal   // Missing quotes--animal is interpreted as a variable,
         // not a string of text
"1.35"   // Numbers in quotes are treated as strings, not numbers
1. 35    // Space before the 3 causes an error
"animal  // Missing closing quotation mark causes an error
```

Automatic Complex Expression Typing

The interpreter computes an expression's value in order to determine its datatype. Consider this example:

```
pointerX = _xmouse;
```

Because `_xmouse` stores the location of the mouse pointer as a number, the type of the expression `_xmouse` will always be a number; so, the variable `pointerX` also becomes a number.

Usually, the datatype that is automatically determined by the interpreter matches what we expect and want. However, some ambiguous cases require us to understand the rules that the interpreter uses to determine an expression's datatype (see Example 2-2 and Example 2-3). Consider the following expression:

```
"1" + 2;
```

The operand on the left of the + is a string ("1"), but the operand on the right is a number (2). The + operator works on both numbers (addition) and strings (concatenation). Should the value of the expression `"1" + 2` be the number 3 or the string "12"? To resolve the ambiguity, the interpreter relies on a fixed rule: the plus operator (+) always favors strings over numbers, so the expression `"1" + 2` evaluates to the string "12", not the number 3. This rule is arbitrary, but it provides a consistent way to interpret the code. The rule was chosen with typical uses of the plus operator in mind: if one of the operands is a string, it's likely that we want to concatenate the operands, not add them numerically, as in this case:

```
trace ("The value of x is: " + x);
```

Combining disparate types of data or using a datum in a context that does not match the expected datatype causes ambiguity. This forces the interpreter to perform an automatic datatype *conversion* according to arbitrary, but predictable, rules. Let's

examine the cases in which automatic conversions will occur and the results of converting a datum from one type to another.

Datatype Conversion

Take a closer look at the example from the previous section. In that example, each datum—"1" and 2—belonged to its own datatype; the first was a string and the second was a number. We saw that the interpreter joined the two values together to form the string "12". Note that the interpreter first had to *convert* the *number* 2 into the *string* "2". Only after that automatic conversion was performed could the value "2" be joined (concatenated) to the string "1".

Datatype conversion simply means changing the type of a datum. Not all datatype conversions are automatic; we may also change a datum's type explicitly in order to override the default datatype conversion that ActionScript would otherwise perform. Explicit conversion is known as *typecasting*, or simply *casting*.

Automatic Type Conversion

Whenever we use a value in a context that does not match the expected datatype, the interpreter attempts a conversion. That is, if the interpreter expects data of type A, and we provide data of type B, the interpreter will attempt to convert our type B data into type A data. For example, in the following code we use the string "Flash" as the right-hand operand of the subtraction operator. Since only numbers may be used with the subtraction operator, the interpreter attempts to convert the string "Flash" into a number:

```
999 - "Flash";
```

Of course, the string "Flash" can't be successfully converted into a legitimate number, so it is converted into the special numeric data value `NaN` (i.e., Not-a-Number). `NaN` is a legal value of the *number* datatype, intended specifically to handle such a situation. With "Flash" converted to `NaN`, our expression ends up looking like this to the interpreter (though we never see this interim step):

```
999 - NaN;
```

Both operands of the subtraction operator are now numbers, so the operation can proceed: `999 - NaN` yields the value `NaN`, which is the final value of our expression.

An expression that yields the numeric value `NaN` isn't particularly useful; most conversions have more functional results. For example, if a string contains only numeric characters, it can be converted into a useful number. The expression:

```
999 - "9";  // The number 999 minus the string "9"
```

is interpreted as:

```
999 - 9;    // The number 999 minus the number 9
```

which yields the value 990 when the expression is resolved. Automatic conversion is most common with the plus operator, the equality operator, the comparison operators, and in conditional or loop statements. In order to be sure of the result of any expression that involves automatic conversion, we have to answer three questions: (a) what is the expected datatype of the current context? (b) what happens when an unexpected datatype is supplied in that context? and (c) when conversion occurs, what is the resulting value?

To answer the first and second questions, we need to consult the appropriate topics elsewhere in this book (e.g., to determine what datatype is expected in a conditional statement, see Chapter 7).

The next three tables, which list the rules of automatic conversion, answer the third question, "When conversion occurs, what is the resulting value?" Table 3-1 shows the results of converting each datatype to a number.

Table 3-1. Converting to a number

Original data	Result after conversion
`undefined`	`0`
`null`	`0`
Boolean	`1` if the original value is `true`; `0` if the original value is `false`
Numeric string	Equivalent numeric value if string is composed only of base-10 numbers, whitespace, exponent, decimal point, plus sign, or minus sign (e.g., "`-1.485e2`")
Other strings	Empty strings, nonnumeric strings, including strings starting with "`x`", "`0x`", or "`FF`", convert to `NaN`
"`Infinity`"	`Infinity`
"`-Infinity`"	`-Infinity`
"`NaN`"	`NaN`
Array	`NaN`
Object	The return value of the object's *valueOf()* method
Movieclip	`NaN`

Table 3-2 shows the results of converting each datatype to a string.

Table 3-2. Converting to a string

Original data	Result after conversion
`undefined`	"" (the empty string)
`null`	"`null`"
Boolean	"`true`" if the original value was `true`; "`false`" if the original value was `false`.
`NaN`	"`NaN`"
`0`	"`0`"
`Infinity`	"`Infinity`"
`-Infinity`	"`-Infinity`"

Table 3-2. Converting to a string (continued)

Original data	Result after conversion
Other numeric value	String equivalent of the number. For example, `944.345` becomes "`944.345`".
Array	A comma-separated list of element values.
Object	The value that results from calling *toString()* on the object. By default, the *toString()* method of an object returns "`[object Object]`". The *toString()* method can be customized to return a more useful result (e.g., *toString()* of a *Date* object returns: "`Sun May 14 11:38:10 EDT 2000`").
Movieclip	The path to the movie clip instance, given in absolute terms starting with the document level in the Player. For example, "`_level0.ball`".

Table 3-3 shows the results of converting each datatype to a Boolean.

Table 3-3. Converting to a Boolean

Original data	Result after conversion
`undefined`	`false`
`null`	`false`
`NaN`	`false`
`0`	`false`
`Infinity`	`true`
`-Infinity`	`true`
Other numeric value	`true`
Nonempty string	`true` if the string can be converted to a valid nonzero number, `false` if not; in ECMA-262, a nonempty string always converts to `true` (Flash diverges from the ECMA standard to maintain compatibility with Flash 4)
Empty string ("")	`false`
Array	`true`
Object	`true`
Movieclip	`true`

Explicit Type Conversion

If the automatic (implicit) type-conversion rules do not suit our purpose, we can manually (explicitly) change a datum's type. When we take matters into our own hands, we must remember that the rules listed in the preceding tables still apply.

Converting to a string with the toString( ) method

We can invoke the *toString()*method to convert any datum to a string. For example:

```
x.toString( );      // Get the string value of the variable x.
(523).toString( );  // Returns "523". Note that we use parentheses
                    // so that the "." isn't treated as a decimal point.
```

When we invoke the *toString()* method on a number, we may also provide a numeric argument indicating the base of the number system in which we'd like the converted string to be represented. This provides a handy means of switching between hexadecimal, decimal, and octal numbers. For example:

```
var myColor = 255;
var hexColor = myColor.toString(16);  // Sets hexColor to "ff"
```

Converting to a string with the String( ) function

The *String()* function has the same result as the *toString()* method, but it uses a different grammar:

```
String(x);    // Convert x to a string
String(523);  // Convert 523 to the string "523"
```

Don't confuse the global *String()* function with the built-in class constructor of the same name. Both are described in the *Language Reference*.

Converting to a string with empty string concatenation

Because the plus operator (+) favors strings in its automatic conversion rules, concatenating the empty string ("") with any datum converts that datum to a string.

```
// Convert x to a string.
x + "";
// Here we check the character position of the number 2 in 523. We first
// concatenate 523 and "", before invoking a String method on the converted value.
trace((523 + "").indexOf(2));
```

Converting to a number with the Number( ) function

Just as the *String()* function converts data to the *string* type, the *Number()* function converts its argument to the *number* type. When conversion to a real number is impossible or illogical, the *Number()* function returns a special numeric value as described in Table 3-1. Here are some examples:

```
Number(age);     // Yields the value of age converted to a number
Number("29");    // Yields the number 29
Number("sara");  // Yields NaN
```

Don't confuse the global *Number()* function with the built-in class constructor of the same name. Both are described in the *Language Reference*.

Because user input in on-screen text fields always belong to the string type, it's necessary to convert text fields to numbers when performing mathematical calculations. For example, if we want to find the sum of the text fields `price1_txt` and `price2_txt`, we use:

```
totalCost = Number(price1_txt.text) + Number(price2_txt.text);
```

Otherwise, `price1_txt` and `price2_txt` will be concatenated as strings, not added as numbers. For more information on text fields, see *TextField* in the *Language Reference*.

Converting to a number by subtracting zero

To trick the interpreter into converting a datum to a number, we can subtract zero from that datum. Again, the conversion follows the rules described in Table 3-1:

```
"953" - 0      // Yields 953
"molly" - 0    // Yields NaN
x - 0          // Yields the value of x converted to a number
```

Converting to a number using the parseInt( ) and parseFloat( ) functions

The *parseInt( )* and *parseFloat( )* functions convert a string containing numbers and letters into a number. The *parseInt( )* function extracts the first integer that appears in a string, provided that the string's first nonblank character is a legal numeric character. Otherwise, *parseInt( )* yields `NaN`. The number extracted via *parseInt( )* starts with the first nonblank character in the string and ends with the character before either the first nonnumeric character or the first occurrence of a decimal point.

Here are some *parseInt( )* examples:

```
parseInt("1a")                // Yields 1
parseInt("1.3a"               // Yields 1
parseInt("     1a")           // Yields 1
parseInt("I am 14 years old") // Yields NaN (the first nonblank
                              // character is not a number)
parseInt("14 years old")      // Yields 14

// Convert decimal to hexadecimal.
(255).toString(16);           // Yields: ff
// Convert hexadecimal to decimal.
parseInt("OxFF");             // Yields 255
```

The *parseFloat( )* function returns the first floating-point number that appears in a string, provided that the string's first nonblank character is a valid numeric character. (A floating-point number is a positive or negative number that contains a decimal value, such as -10.5 or 345.678.) Like *parseInt( )*, *parseFloat( )* yields the special numeric value `NaN` if the string's first nonblank character is not a valid numeric character. The number extracted by *parseFloat( )* is the numeric conversion of the series of characters that starts with the first nonblank character in the string and ends with the character before the first nonnumeric character (any character other than +, –, 0–9, a decimal point, or an *e* or *E* when used for exponential notation).

Here are some *parseFloat( )* examples:

```
parseFloat("1.3a");            // Extracts 1.3
parseFloat("2.75 years old")   // Extracts 2.75
parseFloat("1nce upon a time") // Extracts 1
parseFloat("I'm 3.5 feet tall") // Yields NaN
```

For more information on *parseInt( )* and *parseFloat( )*—including how to specify a *radix* to convert between number systems—see the *Language Reference*.

Converting to a Boolean

When we want to convert a datum to a Boolean, we can use the global *Boolean()* function, which uses similar syntax to the *String()* and *Number()* functions. For example:

```
Boolean(5);  // The result is true
Boolean(x);  // Converts value of x to a Boolean
```

Don't confuse the global *Boolean()* function with the built-in class constructor of the same name. Both are described in the *Language Reference*.

Conversion Duration

All type conversions performed on variables, array elements, and object properties are temporary unless the conversion happens as part of an assignment. Here we see a temporary conversion:

```
var x = "10";     // x is a string.
y = x - 5;        // y is now 5; x's value was temporarily converted to a number.
trace(typeof x);  // Displays: "string"; the conversion was temporary because
                  // it occurred incidentally while evaluating an expression.
```

Here we see a permanent conversion that is the result of an assignment:

```
x = "10";         // x is a string.
x = x - 5;        // x is converted permanently to a number.
trace(typeof x);  // Displays: "number"; the conversion was permanent because
                  // it occurred as part of an assignment.
```

Determining the Type of an Existing Datum

To determine what kind of data is held in a given expression before, say, proceeding with a section of code, we use the *typeof* operator, as follows:

```
typeof expression;
```

The *typeof* operator returns a string telling us the datatype of `expression`, according to Table 3-4.

Table 3-4. Return values of typeof

Original datatype	typeof return value
Number	"number"
String	"string"
Boolean	"boolean"
Object	"object"
Array	"object"
null	"null"
Movieclip	"movieclip"

Table 3-4. Return values of typeof (continued)

Original datatype	typeof return value
Function	"function"
undefined	"undefined"

Here are a few examples:

```
trace(typeof "game over");    // Displays: "string" in the Output window
var x = 5;
trace(typeof x);              // Displays: "number"
var now = new Date();
trace(typeof now);            // Displays: "object"
```

As shown in Example 3-1, when combined with a *for-in* statement, *typeof* provides a handy way to find all the movie clip instances on a timeline. Once the clips are identified, we can assign them to an array for programmatic handling. (If you can't follow all of Example 3-1, revisit it after completing Part I.)

Example 3-1. Populating an array with dynamically identified movie clips

```
// Create an array in which to store the clips.
var childClips = new Array( );

// Check all the properties of the main timeline.
for (prop in _root) {
  // If the current property is a movie clip...
  if (typeof _root[prop] == "movieclip") {
    // ...add it to the clips array.
    childClips.push(_root[prop]);
  }
}

// Now that our array is populated, we can use it to manipulate the clips it contains.
childClips[0]._x = 0;  // Place the first clip on the left of the Stage.
childClips[1]._y = 0;  // Place the second clip at the top of the Stage.
```

Primitive Data Versus Composite Data

So far we've been working mostly with numbers and strings, which are the most common *primitive* datatypes. Primitive datatypes are the basic units of a language; each primitive value represents a single datum (as opposed to an array of multiple items) and holds that datum directly, rather than holding its address elsewhere in memory.

ActionScript supports these primitive datatypes: *number*, *string*, *boolean*, *undefined*, and *null*. ActionScript does not have a separate single-character datatype (e.g., *char*) as found in C/C++ (strings are a primitive datatype in ActionScript, and not arrays of chars as they are in C/C++).