Object-Oriented Development with ActionScript 2.0

Essential ActionScript 2.0

O'REILLY®

Colin Moock

O'REILLY®

Essential ActionScript 2.0

Macromedia Flash, already the de facto standard for delivering multimedia over the Web, is used increasingly to develop web-based applications (so-called Rich Internet Applications). Introduced in Flash MX 2004 and Flash MX Professional 2004, ActionScript 2.0 is a major upgrade to Flash's scripting language, which radically improves object-oriented development in Flash by formalizing objected-oriented programming (OOP) syntax and methodology.

Essential ActionScript 2.0, from the author of the widely acclaimed *ActionScript for Flash MX: The Definitive Guide*, covers not only ActionScript 2.0 syntax, but also object-oriented design and object-oriented programming. This book is targeted at ActionScript developers who want to know how ActionScript 2.0 development differs from ActionScript 1.0, how to upgrade legacy code to ActionScript 2.0, and how to take maximum advantage of ActionScript 2.0 and its OOP features. If you are an experienced OOP developer coming from another language such as Java or C++, *Essential ActionScript 2.0* shows you how to leverage your OOP knowledge in Flash.

Part I teaches object-oriented concepts, syntax, and usage in ActionScript 2.0. It covers strict datatyping, type casting, classes, objects, methods, properties, inheritance, composition, interfaces, classpaths, packages, and exception handling. Beyond teaching mere basics, it helps you to properly design and structure your code.

Part II teaches best practices for setting up and architecting an object-oriented project, plus how user interface components and movie clip subclasses fit into a well-structured Flash application. You'll learn how to structure entire applications and exchange code with other developers to help you build more stable, scalable, and extensible applications.

Part III teaches you to apply proven and widely accepted object-oriented programming strategies—known as design patterns—to Flash. After a brief introduction to design patterns, this section covers the Observer, Singleton, and Model-View-Controller design patterns, plus the delegation event model, with particular attention to their implementation in ActionScript 2.0.

"This book delivers a complete education in harnessing the power of ActionScript 2.0, coupled with the best practices for doing so. Colin illustrates not just how to write ActionScript but how to write great ActionScript."

—Gary Grossman, Flash Architect and Creator of ActionScript, Macromedia

www.oreilly.com

US $39.95 **CAN $57.95**

ISBN-10: 0-596-00652-7

ISBN-13: 978-0-596-00652-5

9 780596 006525 53995

Essential ActionScript 2.0

Colin Moock

O'REILLY®
Beijing • Cambridge • Farnham • Köln • Paris • Sebastopol • Taipei • Tokyo

Essential ActionScript 2.0
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: Sarah Sherman

Cover Designer: Ellie Volckhausen

Interior Designer: David Futato

Printing History:

June 2004: First Edition.

This book uses RepKover™, a durable and flexible lay-flat binding.

ISBN: 0-596-00652-7
[M] [6/06]

to gray, the wonderkid

Table of Contents

Foreword

I came to Macromedia in the summer of 2000, shortly after graduating from college, to start working as a software engineer on the Flash team. In my first days at the company, the team was working tirelessly to ship Flash 5, and everyone was too busy to give me much work to do, let alone guide me in the ways of Macromedia corporate life. Little did I realize that as I was learning my way around the complex C++ architecture of the Flash authoring tool, ActionScript was also beginning its own career in the web development industry. Flash 5 was a landmark release for the Flash authoring tool: it brought ActionScript from an interface that required point-and-click interaction to a full-fledged scripting language based on the ECMAScript standard, with a real text editor. I arrived just as the Flash team was putting real scripting power in the hands of Flash developers. Over the next two releases of Flash, I participated in the continuation of that effort, first by producing the ActionScript debugger in Flash MX and, most recently, by developing the ActionScript 2.0 compiler. My past few years are inextricably linked to this language, and it has contributed to my growth, just as I have contributed to its growth.

In the beginning, my feelings about ActionScript were similar to the feelings a lot of traditional developers have when coming to the language. I found myself comfortable with its flexibility, yet frustrated with its limitations. I was happy to bring features such as the debugger to life, because it helped Flash meet my own expectations of a programming environment. I enjoyed working to close the gaps in Flash's capabilities, feature by feature. With Flash MX, we made strides by greatly improving the code editor and by enabling users to debug their ActionScript. However, ActionScript 1.0 still had one frustrating limitation that we did not address in Flash MX: it was possible to write code that employed object-oriented programming (OOP) techniques, but doing so was complex and unintuitive and not well integrated with Flash concepts like library symbols.

With Flash MX 2004 and ActionScript 2.0, we have arrived at yet another major landmark in ActionScript's evolution. ActionScript 2.0 offers a more sophisticated syntax for the OOP constructs that ActionScript has always supported. ActionScript 2.0 is

easier to learn than its predecessor, and it is closer to other industry-standard programming languages, such as Java and C#. It gives developers the framework needed to build and maintain large, complex applications. In addition, our implementation required minimal changes to the Flash Player, meaning that ActionScript 2.0 can be exported to Flash Player 6, which was already nearly ubiquitous at the time of Flash MX 2004's release.

In the short time that ActionScript has been around, developers have found it to be extraordinarily powerful. Flash places few constraints on the developer's access to the *MovieClip* hierarchy and object model, permitting them to do anything, anywhere. This flexibility has stirred the creativity of our users, enabling them to grow into ActionScript and experiment with it. However, the lack of structure in ActionScript 1.0 made applications difficult to scale up, leading to unwieldy projects that teams found challenging to maintain and organize. It was too easy to write poor code, not to mention place code in locations almost impossible to find by others unfamiliar with the project. ActionScript 2.0 aspires to address these pitfalls by encouraging a structure that all developers can adhere to and understand. Moreover, the ActionScript 2.0 compiler provides developers with feedback on errors that otherwise wouldn't be found until they manifested as bugs at runtime. Still, ActionScript continues to provide extensive and unique control over graphical elements. We strove to ensure that ActionScript is a powerful language moving forward, without treading on the toes of already-seasoned scripters.

ActionScript 2.0 was also the basis for several other notable elements of Flash MX 2004.

The following are all written in ActionScript 2.0:

- The second generation of components (i.e., the v2 components)
- The new Screens metaphor, which includes Slides and Forms (available only in Flash MX Professional 2004)
- The sophisticated data integration capabilities
- The multilingual resource support offered by the Strings panel

Building significant, large-scale features using ActionScript 2.0 provided valuable testing and validation to those of us working on the compiler and informed many of our design decisions. More importantly, these features give Flash developers comprehensive, working examples of ActionScript 2.0 in action (see the *Macromedia/Flash MX 2004/en/First Run/Classes* folder under your application's installation folder). Likewise, the benefits of ActionScript 2.0 are readily apparent in these features, which all consist of classes that are well organized in the *mx.** class hierarchy. In addition, it is easier than ever to determine which code corresponds to the different components, as ActionScript 2.0 has made it possible to eliminate troublesome relics of ActionScript's past, such as the `#initclip` pragma (compiler directive).

ActionScript started life as a few scripting commands inserted by mouse clicks. Five years later, it is a full-featured object-oriented language with which large, complex applications can be developed. Furthermore, it presents a clean, simple syntax that is easy to read and straightforward for a beginner to pick up. In my two releases of the Flash authoring tool, I have learned more and more about ActionScript each step of the way, and now I am proud to have helped redefine it. Colin Moock's previous book, *ActionScript for Flash MX: The Definitive Guide*, was indispensable to me, even as I've worked on the new face of ActionScript. It is the single book you'll find within easy reach at the desk of every engineer on the Flash team. Many of our engineers here were already looking forward to this new book, *Essential ActionScript 2.0*, before it shipped. And with good reason. In this volume, Moock has once again applied his insightful, conversational style to complex topics, teaching not only the syntax of ActionScript 2.0 but also the theory and principles of OOP. He has thoroughly researched the relationships between ActionScript 2.0, its predecessor, and other languages, and he illustrates their differences in precise detail. Moock's intimate familiarity with Flash and ActionScript is evident in this instructive and approachable text, which certainly is an essential companion for anyone wishing to learn and master the ActionScript 2.0 language.

—Rebecca Sun
Senior Software Engineer
Macromedia Flash Team
March 2004

Preface

In September 2003, Macromedia released Flash MX 2004, and, with it, ActionScript 2.0—a drastically enhanced version of Flash's programming language.

ActionScript 2.0 introduces a formal *object-oriented programming* (OOP) syntax and methodology for creating Flash applications. Compared to traditional timeline-based development techniques, ActionScript 2.0's OOP-based development techniques typically make applications:

- More natural to plan and conceptualize
- More stable and bug-free
- More reusable across projects
- Easier to maintain, change, and expand on
- Easier to test
- Easier to codevelop with two or more programmers

Those are some extraordinary qualities. So extraordinary, in fact, that they've turned this book into something of a zealot. This book wants you to embrace ActionScript 2.0 with a passion.

This Book Wants You

This book wants you to use object-oriented programming in your daily Flash work. It wants you to reap the benefits of OOP—one of the most important revolutions in programming history. It wants you to understand ActionScript 2.0 completely. And it will stop at nothing to get what it wants.

Here's its plan...

First, in Part I, *The ActionScript 2.0 Language*, this book teaches you the fundamentals of object-oriented concepts, syntax, and usage. Even if you have never tried object-oriented programming before, Part I will have you understanding and

applying it. Chapter 1 gives an overview of ActionScript 2.0. Chapter 2 teaches you the basics of OOP and helps you decide how much is right for your projects. Chapters 3 through 10 offer details on classes, objects, methods, properties, inheritance, composition, interfaces, packages, and myriad other core OOP concepts. If you already know a lot about OOP because you program in Java or another object-oriented language, this book helps you leverage that prior experience. It draws abundant comparisons between Flash-based OOP and what you already know. Along the way, it introduces OOP into your regular routine through exercises that demonstrate real-world Flash OOP in action.

In Part II, *Application Development*, this book teaches you how to structure entire applications with ActionScript 2.0. In Chapter 11, you'll learn best practices for setting up and architecting an object-oriented project. In Chapters 12 and 13, you'll learn how user interface components and movie clips fit into a well-structured Flash application. In Chapter 14, you'll see how to parcel up and share code with other developers. All this will help you build more scalable, extensible, stable apps. It's all part of this book's plan.

Finally, in Part III, *Design Pattern Examples in ActionScript 2.0*, you'll explore a variety of approaches to various programming situations. You'll see how to apply proven and widely accepted object-oriented programming strategies—known as *design patterns*—to Flash. The design patterns in Part III cover two key topics in Flash development: event broadcasting and user interface management. After an introduction to design patterns in Chapter 15, we'll explore four common patterns in Chapters 16 through 19. Once you've tried working with the patterns presented in Part III, you'll have confidence consulting the larger body of patterns available online and in other literature. And you'll have the skills to draw on other widely recognized object-oriented practices. You see, this book knows it won't be with you forever. It knows it must teach you to find your own solutions.

This book doesn't care whether you already know the meaning of the words "class," "inheritance," "method," "prototype," or "property." If you have no idea what OOP is or why it's worthwhile, this book is delighted to meet you. If, on the other hand, you're already a skilled object-oriented developer, this book wants to make you better. It wants you to have the exhaustive reference material and examples you need to maximize your productivity in ActionScript 2.0.

This book is determined to make you an adept object-oriented programmer. And it's confident it will succeed.

What This Book Is Not

While this book is zealous about core ActionScript 2.0 and object-oriented programming, it does not cover every possible ActionScript-related topic. Specifically, you won't find much discussion of companion technologies, such as Flash Remoting or

Flash Communication Server, nor will you find a dictionary-style Language Reference, as you do in *ActionScript for Flash MX: The Definitive Guide* (O'Reilly). Whereas that book describes the Flash Player's native functions, properties, classes, and objects, this book teaches you how to use those classes and objects, and how to fit them into your own custom-built structures. The built-in library of classes available in the Flash Player changed only incrementally in Flash Player 7, so *ActionScript for Flash MX: The Definitive Guide,* continues to be a worthwhile reference—even to ActionScript 2.0 developers. It makes the perfect companion to *Essential ActionScript 2.0*.

This book does not cover the Screens feature (including Slides and Forms), which is supported only in Flash MX Professional 2004. Screens are used to develop user interfaces visually (in the tradition of Microsoft Visual Basic) and to create slideshow presentations (in the tradition of Microsoft PowerPoint). Although the feature is not a direct topic of study, you'll certainly be prepared to explore Screens on your own once you understand the fundamentals taught by this text.

This book is also not a primer on programming basics, such as conditionals (*if* statements), loops, variables, arrays, and functions. For a gentle introduction to programming basics in Flash, again see *ActionScript for Flash MX: The Definitive Guide*.

Finally, this book does not teach the use of the Flash authoring tool, except as it applies to application development with ActionScript 2.0. For help with the authoring tool, such as creating graphics or timeline animations, you should consult the in-product documentation or any of the fine third-party books available on the topic, including O'Reilly's own *Flash Out of the Box*, by Robert Hoekman, scheduled for release in the second half of 2004.

Who Should (and Shouldn't) Read This Book

You should read this book if you are:

- An intermediate ActionScript 1.0 or JavaScript programmer who understands the basics of variables, loops, conditionals, functions, arrays, and other programming fundamentals.
- An advanced ActionScript 1.0 or ActionScript 2.0 programmer who wants hard facts about best practices for OOP in ActionScript 2.0, including detailed syntax and usage information, language idiosyncrasies, and sample application structures.
- A Flash designer who does some programming and is curious to learn more about application development.
- A programmer migrating to Flash development from another language, such as Java, C++, Perl, JavaScript, or ASP. (Be prepared to learn the fundamentals of the Flash authoring tool from the sources mentioned earlier. You should also read Chapter 13, *Movie Clips*, in *ActionScript for Flash MX: The Definitive Guide*, available online at: *http://moock.org/asdg/samples*.)

You should not read this book if you are a Flash designer/animator with little or no programming experience. (Start your study of ActionScript with *ActionScript for Flash MX: The Definitive Guide* instead.)

ActionScript 2.0 Versus ActionScript 1.0

Chapter 1 introduces ActionScript 2.0 in more detail, but this discussion provides a brief orientation for ActionScript 1.0 developers.

ActionScript 1.0 and ActionScript 2.0 have the same core syntax. Basics like conditionals, loops, operators, and other non-object-oriented aspects of ActionScript 1.0 can be used verbatim in ActionScript 2.0 and are still an official part of the language. In addition, object creation, property access, and method invocation have the same syntax in ActionScript 1.0 and ActionScript 2.0. So, generally speaking, ActionScript 2.0 is familiar to ActionScript 1.0 developers. The main difference between the two versions of the language is object-oriented syntax and authoring tool support for object-oriented development.

In ActionScript 1.0, object-oriented programming had an unintuitive syntax and nearly no authoring tool support (e.g., no compiler messages, no class file structure, no type checking, poor connections between code and movie assets, etc.). With ActionScript 1.0, object-oriented programming was an awkward, esoteric undertaking. With ActionScript 2.0, it is a natural endeavor. ActionScript 2.0's more traditional OOP implementation makes ActionScript 2.0 skills more transferable to and from other languages.

If you're an ActionScript 1.0 programmer and have already been applying OOP techniques, ActionScript 2.0 will be a delight (and a relief) to work with. If you're an ActionScript 1.0 programmer who doesn't use OOP, you don't need to learn OOP in ActionScript 1.0 before you learn it in ActionScript 2.0. Now is the perfect time to explore and adopt this important methodology. OOP offers to increase your productivity, make your projects easier to manage, and improve your code's quality and reusability.

Although this book doesn't spend a lot of time focusing on how to upgrade your code from ActionScript 1.0 to ActionScript 2.0, after reading it, you should have no trouble doing so. The book focuses on giving you a strong fundamental understanding of ActionScript 2.0 and I didn't want to unnecessarily distract from that focus by talking too much about obsolete ActionScript 1.0 code. That said, keep an eye out for the numerous ActionScript 1.0 notes that look like this:

Such notes directly compare an ActionScript 1.0 technique with the analogous ActionScript 2.0 technique, so you can see the difference between the old way of doing things and the new, improved way.

Finally let's be clear about what I mean by "programming in ActionScript 2.0 versus ActionScript 1.0." If you are just creating timeline code and not using ActionScript 2.0 classes, static datatypes, or other OOP features, then it is really moot whether you refer to your code as "ActionScript 1.0" or "ActionScript 2.0." Without using OOP features, ActionScript 2.0 code looks indistinguishable from ActionScript 1.0 code. So when I say, "we're going to learn to program in ActionScript 2.0," of necessity, I'm assuming you're creating a meaningful OOP application in which you're developing one or more classes. For an example, consider an online form that merely sends an email. You might implement that form entirely on the Flash timeline using only variables and functions. If that's generally all you want to do with your applications, then frankly, this book might be overkill for your current needs. However, given the chance, this book will expand your horizons and teach you how to be a skilled object-oriented programmer and to tackle larger projects. So when I say "programming in ActionScript 2.0," I mean "developing object-oriented applications in ActionScript 2.0." The emphasis is on "object-oriented development" rather than ActionScript 2.0, per se, as ActionScript 2.0 is just a means to that end. You may ask, "Is this book about ActionScript 2.0 syntax, object-oriented design, or object-oriented programming?" The answer is, "All of the above."

For more information about ActionScript 2.0 and ActionScript 1.0 in relation to Flash Player 6 and Flash Player 7, see Chapter 1.

Deciphering Flash Versions

With the introduction of the Studio MX family of products, including Flash MX, Macromedia abandoned a standard numeric versioning system for its Flash authoring tool. Subsequent to Flash MX, Macromedia incorporated the year of release in the product name (products released after September use the following year in the product name). With the 2004 release, Macromedia also split the Flash authoring tool into two versions: Flash MX 2004 and Flash MX Professional 2004, as discussed in Table P-1. The principal features specific to the Professional edition are:

- Screens (form- and slide-based content development)
- Additional video tools
- Project and asset management tools
- An external script editor
- Databinding (linking components to data sources obtained via web services, XML, or record sets)
- Advanced components (however, Flash MX Professional 2004 components work happily in Flash MX 2004)
- Mobile device development tools

The techniques taught in this book can be used in both Flash MX 2004 and Flash MX Professional 2004, although I note the rare circumstances in which the two versions differ as pertaining to development in ActionScript 2.0. Unlike the Flash authoring tool, the Flash Player is still versioned numerically; at press time, the latest version is Flash Player 7. Table P-1 describes the naming conventions used in this book for Flash versions.

Table P-1. Flash naming conventions used in this book

Name	Meaning
Flash MX	The version of the Flash authoring tool that was released at the same time as Flash Player 6.
Flash MX 2004	The standard edition of the Flash authoring tool that was released at the same time as Flash Player 7. In the general sense, the term "Flash MX 2004" is used to refer to both the standard edition (Flash MX 2004) and the Professional edition (Flash MX Professional 2004) of the software. When discussing a feature that is limited to the Professional edition, this text states the limitation explicitly.
Flash MX Professional 2004	The Professional edition of the Flash authoring tool that was released at the same time as Flash Player 7. The Professional edition includes some features not found in the standard edition (see preceding list). The Professional edition is not required for this book or to use ActionScript 2.0.
Flash Player 7	The Flash Player, version 7. The Flash Player is a browser plugin for major web browsers on Windows and Macintosh. At press time, Flash Player 6, but not Flash Player 7, was available for Linux. There are both ActiveX control and Netscape-style versions of the plugin, but I refer to them collectively as "Flash Player 7."
Flash Player *x*.0.*y*.0	The Flash Player, specifically, the release specified by major version number *x* and major build number *y*, as in Flash Player 7.0.19.0. The minor version number and minor build number of publicly released versions is always 0.
Standalone Player	A version of the Flash Player that runs directly as an executable 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.

Example Files and Resources

The official companion website for this book is:

http://moock.org/eas2

You can download the example files for this book at:

http://moock.org/eas2/examples

More example Flash code can be found at the Code Depot for *ActionScript for Flash MX: The Definitive Guide*:

http://moock.org/asdg/codedepot

For a long list of Flash-related online resources, see:

http://moock.org/moockmarks

For an extensive collection of links to hundreds of ActionScript 2.0 resources, see:

http://www.actionscripthero.com/adventures

Typographical Conventions

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

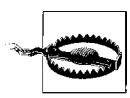
Menu options
: Menu options are shown using the → character, such as File → Open.

`Constant width`
: Indicates code examples, code snippets, clip instance names, frame labels, property names, variable names, and symbol linkage identifiers.

Italic
: Indicates function names, method names, class names, package names, layer names, URLs, filenames, and file suffixes such as *.swf*. In addition to being italicized in the body text, 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.



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

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

This is a note about ActionScript 1.0. It compares and contrasts ActionScript 1.0 with ActionScript 2.0, helping you to migrate to ActionScript 2.0 and to understand important differences between the two versions of the language.

Using Code Examples

This book is here to help you get your job done. In general, you may use the code in this book in your programs and documentation. You do not need to contact us for permission unless you're reproducing a significant portion of the code. For example, writing a program that uses several chunks of code from this book does not require permission. Selling or distributing a CD-ROM of examples from O'Reilly books does require permission. Answering a question by citing this book and quoting example code does not require permission. Incorporating a significant amount of example code from this book into your product's documentation does require permission.

We appreciate, but do not require, attribution. An attribution usually includes the title, author, publisher, and ISBN. For example: "*Essential ActionScript 2.0* by Colin Moock. Copyright 2004 O'Reilly Media, Inc., 0-596-00652-7"

If you feel your use of code examples falls outside fair use or the permission given above, feel free to contact us at permissions@oreilly.com.

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 Media, 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/0596006527

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

Sometimes you're given the opportunity to thank someone but you know you won't be able to fully express the magnitude of your appreciation. You can say what you want, but ultimately you just have to trust that the person knows how deeply grateful you are. I trust that Rebecca Sun, Macromedia's lead ActionScript 2.0 developer, knows.

I'm in a similar boat with Derek Clayton. I've been working with Derek for years on Unity, our commercial framework for creating multiuser applications (see *http://moock.org/unity*). Derek's been a programming mentor to me since I met my first *if* statement, and he's been a friend for even longer. I learn something from him almost every day. This book is filled with the wisdom he has imparted to me over the years.

Bruce Epstein, my editor. What can you say? He is, quite simply, the best. No hyperbole can exaggerate his merit, nor do it justice, so I shall attempt none.

I'd also like to thank all of the members of O'Reilly's editorial, production, interior design, art, marketing, and sales teams including Glenn Bisignani, Claire Cloutier, Colleen Gorman, Tim O'Reilly, Rob Romano, Sarah Sherman, Ellen Troutman, and Ellie Volckhausen. Also my thanks to the copy editor, Norma Emory, for helping to ensure the text's consistency, readability, and accuracy.

Then there are the members of Macromedia's Flash team, who have been a constant source of inspiration, knowledge, and friendship to me since there was a "Flash." I believe that anyone interested in computers is indebted to the whole Flash team for constantly pioneering new forms of computer-based communication. Above all, for his unending support and kindness, I owe Gary Grossman a lifetime of deep bows, obsequious "thank yous," and long handshakes. Specific members of the Flash team, past and present, that I'm honored to know and work with are: Nigel Pegg, Michael Williams, Erica Norton, Waleed Anbar, Deneb Meketa, Matt Wobensmith, Mike Chambers, Chris Thilgen, Gilles Drieu, Nivesh Rajbhandari, Tei Ota, Troy Evans, Lucian Beebe, John Dowdell, Bentley Wolfe, Jeff Mott, Tinic Uro, Robert Tatsumi, Michael Richards, Sharon Seldon, Jody Zhang, Jim Corbett, Karen Cook, Jonathan Gay, Pete Santangeli, Sean Kranzberg, Michael Morris, Kevin Lynch, Ben Chun, Eric Wittman, Jeremy Clark, and Janice Pearce.

I was extraordinarily fortunate to have some truly wonderful technical reviewers and beta readers for this book. Rebecca Sun lent her sage eye to the entire text. Gary Grossman reviewed key sections, including Chapter 10. The following keen beta readers guided me throughout the writing process: Alistair McLoed, Chafic Kazoun, Jon Williams, Marcus Dickinson, Owen Van Dijk, Peter Hall, Ralf Bokelberg, Robert Penner, and Sam Neff. Special thanks to Mark Jonkman and Nick Reeder for their consistently thorough examinations of the manuscript.

Love to my wife, Wendy, who completes me. To my family and friends. And to the trees, for providing the answer to any question, the splendor of any dream, and the paper upon which this book is printed.

—Colin Moock
Toronto, Canada
March 2004

PART I

The ActionScript 2.0 Language

Part I teaches you the fundamentals of object-oriented concepts, syntax, and usage in ActionScript 2.0. Even if you have never tried object-oriented programming before, Part I will have you understanding and applying it. Part I covers classes, objects, methods, properties, inheritance, composition, interfaces, packages, and myriad other core OOP concepts. Beyond teaching you the basics of OOP, it helps you decide how much OOP is right for your projects, and how to structure your classes and their methods.

CHAPTER 1

ActionScript 2.0 Overview

Over the course of this book, we'll study ActionScript 2.0 and object-oriented programming in Flash exhaustively. There's lots to learn ahead but, before we get into too much detail, let's start with a quick summary of ActionScript 2.0's core features and Flash Player 7's new capabilities. If you have an ActionScript 1.0 background, the summary will give you a general sense of what's changed in the language. If, on the other hand, you're completely new to Flash or to ActionScript, you may want to skip directly to Chapter 2.

ActionScript 2.0 Features

Introduced in Flash MX 2004 and Flash MX Professional 2004, ActionScript 2.0 is a major grammatical overhaul of ActionScript as it existed in Flash 5 and Flash MX (retroactively dubbed *ActionScript 1.0*). ActionScript 2.0 adds relatively little new runtime functionality to the language but radically improves object-oriented development in Flash by formalizing objected-oriented programming (OOP) syntax and methodology.

While ActionScript 1.0 could be used in an object-oriented way, it lacked a traditional, official vocabulary for creating classes and objects. ActionScript 2.0 adds syntactic support for traditional object-oriented features. For example, ActionScript 2.0 provides a *class* keyword for creating classes and an *extends* keyword for establishing inheritance. Those keywords were absent from ActionScript 1.0 (though it was still possible to create prototypical objects that could be used as classes). The traditional OOP syntax of ActionScript 2.0 makes the language quite familiar for programmers coming from other OOP languages such as Java and C++.

Most of the new OOP syntax in ActionScript 2.0 is based on the proposed ECMAScript 4 standard. Its specification is posted at *http://www.mozilla.org/js/language/es4*.

Here are some of the key features introduced in ActionScript 2.0. Don't worry if these features are new to you; the remainder of the book covers them in detail:

- The *class* statement, used to create formal classes. The *class* statement is covered in Chapter 4.
- The *extends* keyword, used to establish inheritance. In ActionScript 1.0 inheritance was typically established using the `prototype` property but could also be established via the `__proto__` property. Inheritance is covered in Chapter 6.
- The *interface* statement, used to create Java-style interfaces (i.e., abstract datatypes). Classes provide implementations for interfaces using the *implements* keyword. ActionScript 1.0 did not support interfaces. Interfaces are covered in Chapter 8.
- The official file extension for class files is *.as*. Formerly, classes could be defined in timeline code or in external *.as* files. ActionScript 2.0 now requires classes to be defined in external class files. Class files can be edited in Flash MX Professional 2004's script editor or in an external text editor.
- Formal method-definition syntax, used to create instance methods and class methods in a class body. In ActionScript 1.0, methods were added to a class via the class constructor's `prototype` property. See Chapter 4.
- Formal getter and setter method syntax, which replaces ActionScript 1.0's *Object.addProperty( )* method. See Chapter 4.
- Formal property-definition syntax, used to create instance properties and class properties in a class body. In ActionScript 1.0, instance properties could be added in several ways—via the class constructor's prototype property, in the constructor function, or on each object directly. Furthermore, in ActionScript 1.0, class properties were defined directly on the class constructor function. See Chapter 4.
- The *private* and *public* keywords, used to prevent certain methods and properties from being accessed outside of a class.
- Static typing for variables, properties, parameters, and return values, used to declare the datatype for each item. This eliminates careless errors caused by using the wrong kind of data in the wrong situation. See Chapter 3 for details on type mismatch errors.
- Type casting, used to tell the compiler to treat an object as though it were an instance of another datatype, as is sometimes required when using static typing. See Chapter 3 for details on casting.
- Classpaths, used to define the location of one or more central class repositories. This allows classes to be reused across projects and helps make source files easy to manage. See Chapter 9.
- Exception handling—including the *throw* and *try/catch/finally* statements—used to generate and respond to program errors. See Chapter 10.

- Easy linking between movie clip symbols and ActionScript 2.0 classes via the symbol Linkage properties. This makes *MovieClip* inheritance easier to implement than in ActionScript 1.0, which required the use of `#initclip` and *Object.registerClass( )*. See Chapter 13.

Features Introduced by Flash Player 7

In addition to the ActionScript 2.0 language enhancements, Flash Player 7 introduces some important new classes and capabilities. These are available only to Flash Player 7–format movies playing in Flash Player 7 or later. (For information on export formats, see "Setting a Movie's ActionScript Version and Player Version," later in this chapter.) Although these features are not the direct topic of study in this book, we'll cover a few of them during our exploration of ActionScript 2.0.

The key new features of Flash Player 7 include:

- New array-sorting capabilities
- The *ContextMenu* and *ContextMenuItem* classes for customizing the Flash Player context menu that appears when the user right-clicks (Windows) or Ctrl-clicks (Macintosh) on a Flash movie
- Cross-domain policy files for permitting data and content to be loaded from an external domain
- ID3 v2 tag support for loaded MP3 files
- Mouse wheel support in text fields (Windows only)
- Improved *MovieClip* depth management methods
- The *MovieClipLoader* class for loading movie clips and images
- The *PrintJob* class for printing with greater control than was previously possible
- Support for images in text fields, including flowing text around images
- Improved text metrics (the ability to obtain more accurate measurements of the text in a text field than was possible in Flash Player 6)
- Cascading stylesheet (CSS) support for text fields, allowing the text in a movie to be formatted with a standard CSS stylesheet
- Improved ActionScript runtime performance
- Strict case sensitivity

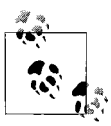
The topic of this book is the core ActionScript 2.0 language. As such, the preceding Flash Player features are not all covered in a detailed manner. For more information on the new features in Flash Player 7, see Flash's online help under Help → ActionScript Reference Guide → What's New in Flash MX 2004 ActionScript.

Flash MX 2004 Version 2 Components

Flash MX introduced *components*—ready-to-use interface widgets and code modules that implement commonly needed functionality. Flash's built-in components make it relatively easy to create desktop-style Flash applications. Flash MX 2004 introduces the new *v2 components*, rewritten from scratch in ActionScript 2.0 and built atop version 2 of the Macromedia Component Architecture, which provides a much richer feature set than its predecessor. The new architecture necessitates new ways of developing and using components (see Chapter 12 for component usage). Officially, the v2 components require Flash Player 6.0.79.0 or higher; however, tests show that many v2 components work in earlier releases of Flash Player 6 (especially Flash Player 6.0.40.0 and higher). If you want to use a v2 component in a version prior to Flash Player 6.0.79.0, you should test your application extensively.

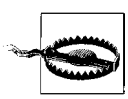
A single application produced in either Flash MX 2004 or Flash MX Professional 2004 can include both v2 components and Flash MX's v1 components, provided the v1 components have been updated to support ActionScript 2.0 and the movie is exported in ActionScript 2.0 format.

Don't confuse v1 and v2 components with the version of ActionScript in which they are written. Granted, v2 components are written in ActionScript 2.0 and there are no ActionScript 1.0 versions of the v2 components. However, although v1 components were written originally in ActionScript 1.0, versions updated to compile under ActionScript 2.0 are available.



The v1 component update for ActionScript 2.0 is available at the Flash Exchange (*http://www.macromedia.com/exchange/flash*), in the User Interface category, under the title "Flash MX Components for Flash MX 2004."

If nonupdated v1 components (i.e., those written in ActionScript 1.0) are used with v2 components in the same movie, some compile-time and runtime errors may occur, depending on the components used.



Do not mix ActionScript 1.0 OOP techniques with ActionScript 2.0 code. If you are using classes, inheritance, and other OOP features, make sure all your code is upgraded to ActionScript 2.0.

Key new v2 component features include:

- A new listener event model for handling component events, which lets many external objects receive a single component's events
- CSS-based stylesheet support, making it easier to change text attributes across components

- Focus management to support tabbing between user interface elements
- Depth management to manage the visual stacking of components on screen
- Richer accessibility support (i.e., better support for screen readers)
- Richer skinning (i.e., graphic replacement) support
- Encapsulation of component assets in a single file, allowing easier component management and sharing

The v2 components tend to be larger than their v1 counterparts. This is especially true if using only one or two components, as the v2 architecture is optimized for applications that use at least three or four different component types. Therefore, if you need only one or two components, and you don't need focus management or accessibility support, you'll get faster (smaller) downloads using the v1 components.

Beware that the default theme ("halo") for the v2 components does not support custom colors for scrollbars and buttons. That is, the `scrollTrackColor` and `buttonColor` style properties do not work with the default v2 component theme in Flash MX 2004 and Flash MX Professional 2004. To set the color of buttons and scrollbars on v2 components, you must apply a new theme to the document. See Help → Using Components → About Themes → Applying a Theme to a Document.

Table 1-1 shows the complete set of components in Flash MX 2004 and Flash MX Professional 2004. Professional components that are not available in Flash MX 2004 will still work in that version of the software. That is, a *.fla* document that contains a component specific to the Professional edition will open normally and work properly in Flash MX 2004. Macromedia's End User License Agreement for Flash MX 2004 does not explicitly prohibit the use of Professional-only components in the standard edition of the software.

Table 1-1. The v1 and v2 components

Component	Flash MX	Flash MX 2004	Flash Pro	Notes
Accordion	[a]		v2	
Alert	[b], [c]		v2	
Button	v1	v2	v2	
CheckBox	v1	v2	v2	
ComboBox	v1	v2	v2	
Data Components			v2	Includes DataHolder, DataSet, RDBMSResolver, WebServiceConnector, XMLConnector, and XUpdateResolver
DataGrid	[b]		v2	
DateChooser	[c], [d]		v2	
DateField			v2	
Label	[a]	v2	v2	

Table 1-1. The v1 and v2 components (continued)

Component	Flash MX	Flash MX 2004	Flash Pro	Notes
List	v1	v2	v2	
Loader	[b]	v2	v2	
Media Components	[b]		v2	MediaController, MediaDisplay, MediaPlayback
Menu	[d]		v2	
MenuBar			v2	
NumericStepper		v2	v2	
ProgressBar	[b, c]	v2	v2	
RadioButton	v1	v2	v2	
ScrollPane	v1	v2	v2	
TextArea	v1	v2	v2	
TextInput	v1	v2	v2	
Tree	[c]		v2	
Window	[c]	v2	v2	

[a] Similar component available in DRK3 (http://www.macromedia.com/software/drk/productinfo/product_overview/volume3).
[b] Similar component available in DRK1 (http://www.macromedia.com/software/drk/productinfo/product_overview/volume1).
[c] Similar component available in Flash UI Component Set 2 at the Flash Exchange (http://www.macromedia.com/exchange/flash).
[d] Similar component available in DRK2 (http://www.macromedia.com/software/drk/productinfo/product_overview/volume2).

In Chapter 12, we'll learn how to program graphical, OOP applications that use the v2 components.

ActionScript 1.0 and 2.0 in Flash Player 6 and 7

ActionScript 1.0 is based on the ECMAScript 3 standard (as is JavaScript 1.5), whereas ActionScript 2.0 is based on the emerging ECMAScript 4 standard (as is the theoretical JavaScript 2.0). As we learned in the Preface, under "ActionScript 2.0 Versus ActionScript 1.0," this common heritage gives the two versions a strong family resemblance; they share the same syntax for most non-OOP features, such as loops, conditionals, and operators.

Although ActionScript 2.0 is now the preferred version of ActionScript, ActionScript 1.0 syntax continues to be fully supported by Flash Player 7 and is not deprecated. As we'll see shortly, you can author either ActionScript 1.0 or ActionScript 2.0 in Flash MX 2004 and Flash MX Professional 2004 (but you cannot author ActionScript 2.0 in Flash MX). With a few minor exceptions, noted throughout the text, ActionScript 2.0 code is also backward compatible with Flash Player 6. However, ActionScript 2.0 is not compatible with older versions such as Flash Player 5 or Flash Player 4.

If you're an ActionScript 1.0 programmer, you can think of ActionScript 2.0 as a syntactic façade over ActionScript 1.0. That is, both ActionScript 2.0 and ActionScript 1.0

compile to the same *.swf* bytecode (with a few minor additions for ActionScript 2.0). To the Flash Player, at runtime, there's effectively no difference between ActionScript 1.0 and ActionScript 2.0 (barring the aforementioned minor additions). For example, once an ActionScript 2.0 class, such as *Rectangle*, is compiled to a *.swf* file, it exists as a *Function* object at runtime, just as an older ActionScript 1.0 function declaration used as a class constructor would. Similarly, at runtime, an ActionScript 2.0 *Rectangle* instance (`r`) is given a `__proto__` property that refers to `Rectangle.prototype`, again making it look to the Flash Player just like its ActionScript 1.0 counterpart.

But for the most part, you don't need to worry about these behind-the-scenes compiler and runtime issues. If you're moving to ActionScript 2.0 (and I think you should!), you can permanently forget ActionScript 1.0's prototype-based programming. In fact, most ActionScript 1.0 techniques for dynamically manipulating objects and classes at runtime are considered bad practice in ActionScript 2.0, and will actually lead to compiler errors when mixed with ActionScript 2.0 code. But never fear, this book highlights problematic ActionScript 1.0 practices and show you how to replace them with their modern ActionScript 2.0 counterparts.

Setting a Movie's ActionScript Version and Player Version

Flash MX 2004 lets you export *.swf* files (a.k.a. movies) in a format compatible with specific versions of the Flash Player. Don't confuse this with the version of the Flash Player the end user has installed (which is beyond your control except for checking their Player version and suggesting they upgrade when appropriate).

To set the version of a *.swf* file, use the drop-down list under File → Publish Settings → Flash → Version. For maximum compatibility, always set your *.swf* file's Flash Player version explicitly to the lowest version required, and no higher. If the *.swf* file version is higher than the end user's version of the Flash Player, it might not display correctly, and most code execution will fail.

Setting the version of a Flash movie has the following effects:

- The movie will be compatible with (i.e., playable in) the specified version of the Flash Player (or later versions). In earlier versions, most ActionScript code will either not execute properly or not execute at all.
- The movie will play properly in the most recent version of the Flash Player, even if it uses features that have changed since the specified version was released. In other words, the newest Flash Player will always play older format *.swf* files properly. For example, ActionScript identifiers in a Flash Player 6–format *.swf* file playing in Flash Player 7 are *not* case sensitive, even though identifiers in Flash Player 7–format *.swf* files *are* case sensitive. However, there's one exception: the security changes to the rules of cross-domain data loading in Flash Player 7 affect Flash Player 6–format *.swf* files in some cases. For details see *http://moock.org/asdg/technotes/crossDomainPolicyFiles*.

When exporting Flash Player 6– and Flash Player 7–format movies from either Flash MX 2004 or Flash MX Professional 2004, you can tell Flash whether to compile your code as if it is ActionScript 1.0 or ActionScript 2.0. Naturally, you should make this choice at the beginning of development, as you don't want to rewrite your code at the end. To specify which version of the ActionScript compiler to use when creating a *.swf* file, use the drop-down list under File → Publish Settings → Flash → ActionScript Version.

Throughout the remainder of the text, this book assumes you are using ActionScript 2.0's compiler.

When the ActionScript version is set to ActionScript 1.0, the following changes take effect:

- ActionScript 2.0 syntax is not recognized and ActionScript 2.0 features, such as type checking (including post-colon syntax) and error handling, can either cause compiler errors (for Flash Player 6–format movies) or simply fail silently (for Flash Player 7–format movies).
- Flash 4–style "slash syntax" for variables is allowed (but this coding style is deprecated and not recommended).
- Reserved words added in ActionScript 2.0 such as *class*, *interface*, and *public* can be used as identifiers (but this practice makes code difficult to update and is highly discouraged).

The following runtime features of ActionScript 2.0 will not work in *.swf* files exported to a Flash Player 6–format *.swf* file, no matter which version of the Flash Player is used:

- Exception handling (see Chapter 10).
- Case sensitivity. (Scripts exported in Flash Player 6–format *.swf* files are not case sensitive, even in Flash Player 7. But beware! ActionScript 1.0 code in a Flash Player 7–format *.swf* file is case sensitive when played in Flash Player 7. See Table 1-2.)
- Type casting (see "Runtime Casting Support" in Chapter 3).

Table 1-2 outlines case sensitivity for various possible permutations of the *.swf* file version and the user's Flash Player version. Note that runtime case sensitivity is unrelated to the ActionScript compiler version chosen and is dependent only on the format of the exported *.swf* file and the Flash Player version. In other words, both ActionScript 1.0 and 2.0 are case sensitive when exported in Flash Player 7–format *.swf* files and played in Flash Player 7. In other cases, code is case insensitive subject to the exceptions cited in the footnotes to Table 1-2. Consult my book *ActionScript for Flash MX: The Definitive Guide* (O'Reilly) for a full discussion of case sensitivity and its implications in Flash Player 6.

Table 1-2. Runtime case sensitivity support by language, file format, and Flash Player version

Movie compiled as either ActionScript 1.0 or 2.0 and	Played in Flash Player 6	Played in Flash Player 7
Flash Player 6–format *.swf* file	Case insensitive[a]	Case-insensitive[a]
Flash Player 7–format *.swf* file	Not supported[b]	Case-sensitive

[a] Identifiers (i.e., variable and property names), function names, frame labels, and symbol export IDs are case insensitive in Flash Player 6–format *.swf* files. However, reserved words such as "if" are case sensitive, even in Flash Player 6.

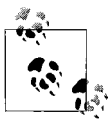
[b] Flash Player 6 cannot play Flash Player 7–format *.swf* files.

Changes to ActionScript 1.0 in Flash Player 7

In a Flash Player 7–format *.swf* file running in Flash Player 7, some ActionScript 1.0 code behaves differently than it does in Flash Player 6. These changes bring Flash Player 7 closer to full ECMAScript 3 compliance. Specifically:

- The value `undefined` converts to the number `NaN` when used in a numeric context and to the string "undefined" when used in a string context (in Flash Player 6, `undefined` converts to the number 0 and to the empty string, "").
- Any nonempty string converts to the Boolean value `true` when used in a Boolean context (in Flash Player 6, a string converts to `true` only if it can be converted to a valid nonzero number; otherwise, it converts to `false`).
- Identifiers (function names, variable names, property names, etc.) are case sensitive. For example, the identifiers `firstName` and `firstname` refer to two different variables in Flash Player 7. In Flash Player 6, the identifiers would refer to a single variable. (However, as usual, frame labels and symbol linkage IDs are not case sensitive.)

The preceding changes affect you only when you are updating a Flash Player 6–format movie to a Flash Player 7–format movie in order to use a feature unique to Flash Player 7. That is, if you upgrade your movie, you must test and possibly modify your code to make sure that it operates the same in Flash Player 7 format as it did in Flash Player 6 format. If you do not need Flash Player 7 features in your movie, you can continue to export it to Flash Player 6 format and it will usually run in Flash Player 7 exactly as it did in Flash Player 6. This last point cannot be emphasized enough.



Macromedia goes to great lengths to ensure that movies exported in older versions of the *.swf* format, such as Flash Player 6 format, continue to operate unchanged even if played in a later Player, such as Flash Player 7. However, when you publish a movie in Flash Player 7 format, you must be mindful of the changes implemented since the previous version of the *.swf* format. That is, the changes needed in your ActionScript depend on the *.swf* file version, not the Flash Player version.

Of course, any newly created *.swf* files exported in Flash Player 7 format (and not just those upgraded from Flash Player 6–format *.swf* files) must obey the new conventions, so keep them in mind moving forward. Remember that these new conventions bring ActionScript in line with other languages such as JavaScript and Java, making it easier to port code to or from other languages.

Flash 4 Slash Syntax Is Not Supported in ActionScript 2.0

In Flash 4 and subsequent versions, variables could be referenced with so-called "slash syntax." For example, in Flash 4, the following code is a reference to the variable `x` on the movie clip `ball`:

```
/ball:x
```

That syntax generates the following error if you attempt to use it with the ActionScript 2.0 compiler, whether exporting in Flash Player 6 or Flash Player 7 format:

```
Unexpected '/' encountered
```

Let's Go OOP

Now that we've had a taste of what ActionScript 2.0 has to offer, we can start our study of object-oriented programming with Flash in earnest. When you're ready to get your hands dirty, move on to Chapter 2!

CHAPTER 2

Object-Oriented ActionScript

Ironically, Flash users who are new to object-oriented programming (OOP) are often familiar with many object-oriented concepts without knowing their formal names. This chapter demystifies some of the terminology and brings newer programmers up to speed on key OOP concepts. It also serves as a high-level overview of OOP in ActionScript for experienced programmers who are making their first foray into Flash development.

Procedural Programming and Object-Oriented Programming

Traditional programming consists of various instructions grouped into *procedures*. Procedures perform a specific task without any knowledge of or concern for the larger program. For example, a procedure might perform a calculation and return the result. In a procedural-style Flash program, repeated tasks are stored in *functions* and data is stored in *variables*. The program runs by executing functions and changing variable values, typically for the purpose of handling input and generating output. Procedural programming is sensible for certain applications; however, as applications become larger or more complex and the interactions between procedures (and the programmers who use them) become more numerous, procedural programs can become unwieldy. They can be hard to maintain, hard to debug, and hard to upgrade.

Object-oriented programming (OOP) is a different approach to programming, intended to solve some of the development and maintenance problems commonly associated with large procedural programs. OOP is designed to make complex applications more manageable by breaking them down into self-contained, interacting modules. OOP lets us translate abstract concepts and tangible real-world things into corresponding parts of a program (the "objects" of OOP). It's also designed to let an application create and manage more than one of something, as is often required by user interfaces. For example, we might need 20 cars in a simulation, 2 players in a game, or 4 checkboxes in a fill-in form.

Properly applied, OOP adds a level of conceptual organization to a program. It groups related functions and variables together into separate *classes*, each of which is a self-contained part of the program with its own responsibilities. Classes are used to create individual objects that execute functions and set variables on one another, producing the program's behavior. Organizing the code into classes makes it easier to create a program that maps well to real-world problems with real-world components. Parts II and III of this book cover some of the common situations you'll encounter in ActionScript, and show how to apply OOP solutions to them. But before we explore applied situations, let's briefly consider the basic concepts of OOP.

Key Object-Oriented Programming Concepts

An *object* is a self-contained software module that contains related functions (called its *methods*) and variables (called its *properties*). Individual objects are created from *classes*, which provide the blueprint for an object's methods and properties. That is, a class is the template from which an object is made. Classes can represent theoretical concepts, such as a timer, or physical entities in a program, such as a pull-down menu or a spaceship. A single class can be used to generate any number of objects, each with the same general structure, somewhat as a single recipe can be used to bake any number of muffins. For example, an OOP space fighting game might have 20 individual *SpaceShip* objects on screen at one time, all created from a single *SpaceShip* class. Similarly, the game might have one *2dVector* class that represents a mathematical vector but thousands of *2dVector* objects in the game.

The term *instance* is often used as a synonym for *object*. For example, the phrases "Make a new *SpaceShip* instance" and "Make a new *SpaceShip* object" mean the same thing. Creating a new object from a class is sometimes called *instantiation*.

To build an object-oriented program, we:

1. Create one or more classes.
2. Make (i.e., *instantiate*) objects from those classes.
3. Tell the objects what to do.

What the objects *do* determines the behavior of the program.

In addition to using the classes we create, a program can use any of the classes built into the Flash Player. For example, a program can use the built-in *Sound* class to create *Sound* objects. An individual *Sound* object represents and controls a single sound or a group of sounds. Its *setVolume( )* method can raise or lower the volume of a sound. Its *loadSound( )* method can retrieve and play an MP3 sound file. And its `duration` property can tell us the length of the loaded sound, in milliseconds. Together, the built-in classes and our custom classes form the basic building blocks of all OOP applications in Flash.

Class Syntax

Let's jump right into a tangible example. Earlier, I suggested that a space fighting game would have a *SpaceShip* class. The ActionScript that defines the class might look like the source code shown in Example 2-1 (don't worry if much of this code is new to you; we'll study it in detail in the coming chapters).

Example 2-1. The SpaceShip class

```
class SpaceShip {
  // This is a public property named speed.
  public var speed:Number;

  // This is a private property named damage.
  private var damage:Number;

  // This is a constructor function, which initializes
  // each SpaceShip instance.
  public function SpaceShip ( ) {
    speed = 100;
    damage = 0;
  }

  // This is a public method named fireMissile( ).
  public function fireMissile ( ):Void {
    // Code that fires a missile goes here.
  }

  // This is a public method named thrust( ).
  public function thrust ( ):Void {
    // Code that propels the ship goes here.
  }
}
```

Notice how the *SpaceShip* class groups related aspects of the program neatly together (as do all classes). Variables (properties), such as `speed` and `damage`, related to spaceships are grouped with functions (methods) used to move a spaceship and fire its weapons. Other aspects of the program, such as keeping score and drawing the background graphics can be kept separate, in their own classes (not shown in this example).

Object Creation

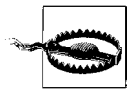
Objects are created (*instantiated*) with the *new* operator, as in:

```
new ClassName( )
```

where `ClassName` is the name of the class from which the object will be created.

For example, when we want to create a new *SpaceShip* object in our hypothetical game, we use this code:

```
new SpaceShip( )
```



The syntax for creating objects (e.g., *new SpaceShip( )*) is the same in ActionScript 2.0 as it was in ActionScript 1.0. However, the syntax for defining classes in ActionScript 2.0 differs from ActionScript 1.0.

Most objects are stored somewhere after they're created so that they can be used later in the program. For example, we might store a *SpaceShip* instance in a variable named `ship`, like this:

```
var ship:SpaceShip = new SpaceShip( );
```

Each object is a discrete data value that can be stored in a variable, an array element, or even a property of another object. For example, if you create 20 alien spaceships, you would ordinarily store references to the 20 *SpaceShip* objects in a single array. This allows you to easily manipulate multiple objects by cycling through the array and, say, invoking a method of the *SpaceShip* class on each object.

Object Usage

An object's methods provide its capabilities (i.e., behaviors)—things like "fire missile," "move," and "scroll down." An object's properties store its data, which describes its state at any given point in time. For example, at a particular point in a game, our `ship`'s current state might be `speed` is 300, `damage` is 14.

Methods and properties that are defined as *public* by an object's class can be accessed from anywhere in a program. By contrast, methods and properties defined as *private* can be used only within the source code of the class or its subclasses. As we'll learn in Chapter 4, methods and properties should be defined as *public* only if they must be accessed externally.

To invoke a method, we use the dot operator (i.e., a period) and the function call operator (i.e., parentheses). For example:

```
// Invoke the ship object's fireMissile( ) method.
ship.fireMissile( );
```

To set a property, we use the dot operator and an equals sign. For example:

```
// Set the ship's speed property to 120.
ship.speed = 120;
```

To retrieve a property's value, we use the dot operator on its own. For example:

```
// Display the value of the speed property in the Output panel.
trace(ship.speed);
```

Encapsulation

Objects are said to *encapsulate* their property values and method source code from the rest of the program. If properly designed, an object's private properties and the internal code used in its methods (including public methods) are its own business;

they can change without necessitating changes in the rest of the program. As long as the method names (and their parameters and return values) stay the same, the rest of the program can continue to use the object without being rewritten.

Encapsulation is an important aspect of object-oriented design because it allows different programmers to work on different classes independently. As long as they agree on the names of the public methods through which they'll communicate, the classes can be developed independently. Furthermore, by developing a specification that shows the publicly available methods, the parameters they require, and the values they return, a class can be tested thoroughly before being deployed. The same test code can be used to reverify the class's operation even if the code within the class is *refactored* (i.e., rewritten to enhance performance or to simplify the source code without changing the previously existing functionality).

In Chapter 4, we'll learn how to use the *private* modifier to prevent a method or property from being accessed by other parts of a program.

Datatypes

Each class in an object-oriented program can be thought of as defining a unique kind of data, which is formally represented as a *datatype* in the program.

A class effectively defines a custom datatype.

You are probably already familiar with custom datatypes defined by built-in ActionScript classes, such as the *Date* class. That is, when you create a *Date* object using *new Date( )*, the returned value contains not a string or a number but a complex datatype that defines a particular day of a particular year. As such, the *Date* datatype supports various properties and methods uniquely associated with dates.

Datatypes are used to impose limits on what can be stored in a variable, used as a parameter, or passed as a return value. For example, when we defined the `speed` property earlier, we also specified its datatype as *Number* (as shown in bold):

```
// The expression ":Number" defines speed's datatype.
public var speed:Number;
```

Attempts to store a nonnumeric value in the `speed` property generate a compile-time error.

If you test a movie and Flash's Output panel displays an error containing the phrase "Type mismatch," you know that you used the wrong kind of data somewhere in your program (the compiler will tell you precisely where). Datatypes help us guarantee that a program isn't used in unintended ways. For example, by specifying that the datatype of `speed` is a number, we prevent someone from unintentionally setting

speed to, say, the string "very fast." The following code generates a compile-time error due to the datatype mismatch:

```
public var speed:Number = "very fast";  // Error!
                                        // You can't assign a String to a
                                        // variable whose type is Number.
```

We'll talk more about datatypes and type mismatches in Chapter 3.

Inheritance

When developing an object-oriented application, we can use *inheritance* to allow one class to adopt the method and property definitions of another. Using inheritance, we can structure an application hierarchically so that many classes can reuse the features of a single class. For example, specific *Car*, *Boat*, and *Plane* classes could reuse the features of a generic *Vehicle* class, thus reducing redundancy in the application. Less redundancy means less code to write and test. Furthermore, it makes code easier to change—for example, updating a movement algorithm in a single class is easier and less error prone than updating it across several classes.

A class that inherits properties and methods from another class is called a *subclass*. The class from which a subclass inherits properties and methods is called the subclass's *superclass*. Naturally, a subclass can define its own properties and methods in addition to those it inherits from its superclass. A single superclass can have more than one subclass, but a single subclass can have only one superclass (although it can also inherit from its superclass's superclass, if any). We'll cover inheritance in detail in Chapter 6.

Packages

In a large application, we can create *packages* to contain groups of classes. A package lets us organize classes into logical groups and prevents naming conflicts between classes. This is particularly useful when components and third-party class libraries are involved. For example, Flash MX 2004's GUI components, including one named *Button*, reside in a package named *mx.controls*. The GUI component class named *Button* would be confused with Flash's built-in *Button* class if it weren't identified as part of the *mx.controls* package. Physically, packages are directories that are collections of class files (i.e., collections of *.as* files).

We'll learn about preventing naming conflicts by referring to classes within a package, and much more, in Chapter 9.

Compilation

When an OOP application is exported as a Flash movie (i.e., a *.swf* file), each class is *compiled*; that is, the compiler attempts to convert each class from source code to

bytecode—instructions that the Flash Player can understand and execute. If a class contains errors, compilation fails and the Flash compiler displays the errors in the Output panel in the Flash authoring tool. The error messages, such as the datatype mismatch error described earlier, should help you diagnose and solve the problem. Even if the movie compiles successfully, errors may still occur while a program is running; these are called *runtime errors*. We'll learn about Player-generated runtime errors and program-generated runtime errors in Chapter 10.

Starting an Objected-Oriented Application

In our brief overview of OOP in Flash, we've seen that an object-oriented application is made up of classes and objects. But we haven't learned how to actually start the application running. Every Flash application, no matter how many classes or external assets it contains, starts life as a single *.swf* file loaded into the Flash Player. When the Flash Player loads a new *.swf* file, it executes the actions on frame 1 and then displays the contents of frame 1.

Hence, in the simplest case, we can create an object-oriented Flash application and start it as follows:

1. Create one or more classes in *.as* files.
2. Create a *.fla* file.
3. On frame 1 of the *.fla* file, add code that creates an object of a class.
4. Optionally invoke a method on the object to start the application.
5. Export a *.swf* file from the *.fla* file.
6. Load the *.swf* file into the Flash Player.

We'll study more complex ways to structure and run object-oriented Flash applications in Chapters 5, 11, and 12.

But How Do I Apply OOP?

Many people learn the basics of OOP only to say, "I understand the terminology and concepts, but I have no idea how or when to use them." If you have that feeling, don't worry, it's perfectly normal; in fact, it means you're ready to move on to the next phase of your learning—*object-oriented design* (OOD).

The core concepts of OOP (classes, objects, methods, properties, etc.) are only tools. The real challenge is designing what you want to build with those tools. Once you understand a hammer, nails, and wood, you still have to draw a blueprint before you can actually build a fence, a room, or a chair. Object-oriented design is the "draw a blueprint" phase of object-oriented programming, during which you organize your entire application as a series of classes. Breaking up a program into classes is a fundamental design problem that you'll face daily in your OOP work. We'll return to this important aspect of OOP regularly throughout this book.

But not all Flash applications need to be purely object-oriented. Flash supports both procedural and object-oriented programming and allows you to combine both approaches in a single Flash movie. In some cases, it's sensible to apply OOP only to a single part of your project. Perhaps you're building a web site with a section that displays photographs. You don't have to make the whole site object-oriented; you can just use OOP to create the photograph-display module. (In fact, we'll do just that in Chapters 5 and 7!)

So if Flash supports both procedural and object-oriented programming, how much of each is right for your project? To best answer that question, we first need to understand the basic structure of every Flash movie. The fundamental organizing structure of a Flash document (a *.fla* file) is the *timeline*, which contains one or more *frames*. Each frame defines the content that is displayed on the graphical canvas called the *Stage*. In the Flash Player, frames are displayed one at a time in a linear sequence, producing an animated effect—exactly like the frames in a filmstrip.

At one end of the development spectrum, Flash's timeline is often used for interactive animation and motion graphics. In this development style, code is mixed directly with timeline frames and graphical content. For example, a movie might display a 25-frame animation, then stop, calculate some random feature used to display another animation, and then stop again and ask the user to fill in a form while yet another animation plays in the background. That is, for simple applications, different frames in the timeline can be used to represent different program *states* (each state is simply one of the possible places, either physical or conceptual, that a user can be in the program). For example, one frame might represent the welcome screen, another frame might represent the data entry screen, a third frame might represent an error screen or exit screen, and so on. Of course, if the application includes animation, each program state might be represented by a range of frames instead of a single frame. For example, the welcome screen might include a looping animation.

When developing content that is heavily dependent on motion graphics, using the timeline makes sense because it allows for precise, visual control over graphic elements. In this style of development, code is commonly attached to the frames of the timeline using the Actions panel (F9). The code on a frame is executed immediately before the frame's content is displayed. Code can also be attached directly to the graphical components on stage. For example, a button can contain code that governs what happens when it is clicked.

Timeline-based development usually goes hand-in-hand with procedural programming because you want to take certain actions at the time a particular frame is reached. In Flash, "procedural programming" means executing code, defining functions, and setting variables on frames in a document's timeline or on graphical components on stage.

However, not all Flash content necessarily involves timeline-based motion. If you are creating a video game, it becomes impossible to position the monsters and the

player's character using the timeline. Likewise, you don't know exactly when the user is going to shoot the monster or take some other action. Therefore, you must use ActionScript instead of the timeline to position the characters in response to user actions (or in response to an algorithm that controls the monsters in some semi-intelligent way). Instead of a timeline-based project containing predetermined animated sequences, we have a nonlinear project in which characters and their behavior are represented entirely in code.

This type of development lends itself naturally to objects that represent, say, the player's character or the monsters. At this end of the development spectrum lies traditional object-oriented programming, in which an application exists as a group of classes. In a pure object-oriented Flash application, a *.fla* file might contain only a single frame, which simply loads the application's main class and starts the application by invoking a method on that main class. Of course, OOP is good for more than just video games. For example, a Flash-based rental car reservation system might have no timeline code whatsoever and create all user interface elements from within classes.

Most real-world Flash applications lie somewhere between the extreme poles of timeline-only development and pure OOP development. For example, consider a Flash-based web site in which two buttons slide into the center of the screen and offer the user a choice of languages: "English" or "French." The user clicks the preferred language button, and both buttons slide off screen. An animated sequence then displays company information and a video showing a product demo. The video is controlled by a *MediaPlayback* component.

Our hypothetical web site includes both procedural programming and OOP, as follows:

- Frames 2 and 3 contain preloader code.
- Frame 10 contains code to start the button-slide animation.
- Frames 11–25 contain the button-slide animation.
- Frame 25 contains code to define button event handlers, which load a language-specific movie.
- In the loaded language-specific movie, frame 1 contains code to control the *MediaPlayback* component.

In the preceding example, code placed directly on frames (e.g., the preloader code) is procedural. But the buttons and *MediaPlayback* component are objects derived from classes stored in external *.as* files. Controlling them requires object-oriented programming. And, interestingly enough, Flash components are, themselves, movie clips. Movie clips, so intrinsic to Flash, can be thought of as self-contained objects with their own timelines and frames. Components (indeed, any movie clip) can contain procedural code internally on their own frames even though they are objects. Such is the nature of Flash development—assets containing procedural code can be mixed on multiple levels with object-oriented code.

As mentioned in the Preface, this book assumes you understand movie clips and have used them in your work. If you are a programmer coming to Flash from another language, and you need a crash course on movie clips from a programmer's perspective, consult Chapter 13 of *ActionScript for Flash MX: The Definitive Guide* (O'Reilly), available online at *http://moock.org/asdg/samples*.

Flash's ability to combine procedural and object-oriented code in a graphical, time-based development environment makes it uniquely flexible. That flexibility is both powerful and dangerous. On one hand, animations and interface transitions that are trivial in Flash might require hours of custom coding in languages such as C++ or Java. But on the other hand, code that is attached to frames on timelines or components on the Stage is time-consuming to find and modify. So overuse of timeline code in Flash can quickly (and quietly!) turn a project into an unmaintainable mess. Object-oriented techniques stress separation of code from assets such as graphics and sound, allowing an object-oriented application to be changed, reused, and expanded on more easily than a comparable timeline-based program. If you find yourself in the middle of a timeline-based project faced with a change and dreading the work involved, chances are the project should have been developed with object-oriented principles from the outset. Although OOP may appear to require additional up-front development time, for most nontrivial projects, you'll recoup that time investment many times over later in the project.

Ultimately, the amount of OOP you end up using in your work is a personal decision that will vary according to your experience and project requirements. You can use the following list to help decide when to use OOP and when to use procedural timeline code. Bear in mind, however, that these are just guidelines—there's always more than one way to create an application. Ultimately, if the software works and can be maintained, you're doing something right.

Consider using OOP when creating:

- Traditional desktop-style applications with few transitions and standardized user interfaces
- Applications that include server-side logic
- Functionality that is reused across multiple projects
- Components
- Games
- Highly customized user interfaces that include complex visual transitions

Consider using procedural programming when creating:

- Animations with small scripts that control flow or basic interactivity

- Simple applications such as a one-page product order form or a newsletter subscription form
- Highly customized user interfaces that include complex visual transitions

You'll notice that the bulleted item "Highly customized user interfaces that include complex visual transitions" is included as a case in which you might use both OOP and procedural programming. Both disciplines can effectively create that kind of content. However, remember that OOP in Flash is typically more maintainable than timeline code and is easier to integrate into version control systems and other external production tools. If you suspect that your highly customized UI will be used for more than a single project, you should strongly consider developing it as a reusable class library or set of components with OOP.

Note that in addition to Flash's traditional timeline metaphor, Flash MX Professional 2004 introduced a *Screens* feature (which includes both *Slides* and *Forms*). Screens provide a facade over the traditional timeline metaphor. Slides and Forms are akin to the card-based metaphors of programs like HyperCard. Slides are intended for PowerPoint-style slide presentations, while Forms are intended for VB developers used to working on multipage forms. Like timeline-based applications, Screens-based applications include both object-oriented code (i.e., code in classes) and procedural-style code (i.e., code placed visually on components and on the Screens of the application). As mentioned in the Preface, this book does not cover Screens in detail, but the foundational OOP skills you'll learn in this text will more than equip you for your own exploration of Screens.

On with the Show!

In this chapter, we summarized the core concepts of OOP in Flash. We're now ready to move on with the rest of Part I, where we'll study all of those concepts again in detail, applying them to practical situations along the way. If you're already quite comfortable with OOP and want to dive into some examples, see Chapters 5, 7, 11, and 12, and all of Part III, which contain in-depth studies of real-world object-oriented code.

Let's get started!

CHAPTER 3
Datatypes and Type Checking

ActionScript 2.0 defines a wide variety of datatypes. Some datatypes are native to the language itself (e.g., *String*, *Number*, and *Boolean*). Others are included in the Flash Player and are available throughout all Flash movies (e.g., *Color*, *Date*, and *TextField*). Still other datatypes are defined by components that can be added individually to Flash movies (e.g., *List*, *RadioButton*, and *ScrollPane*).

For a primer on ActionScript's datatypes, see Chapter 3 of *ActionScript for Flash MX: The Definitive Guide* (O'Reilly), available online at *http://moock.org/asdg/samples*.

In addition to using ActionScript 2.0's datatypes, developers can add new datatypes to a program by creating classes (covered in Chapter 4) and interfaces (covered in Chapter 8). Every value in ActionScript 2.0 belongs to a datatype, whether built-in or programmer-defined. When we work with a value, we must use it only in ways supported by its datatype. For example, we can call *getTime( )* on a *Date* object, but we must not call *gotoAndPlay( )* on a *Date* object, because the *Date* class does not support the *gotoAndPlay( )* method. On the other hand, we can call *gotoAndPlay( )* on a movie clip because that method is defined by the *MovieClip* class.

In order for an object-oriented program to work properly, every operation performed on every object should succeed. That is, if a method is invoked on an object, the object's class must actually define that method. And if a property is accessed on an object, the object's class must define that property. If the object's class does not support the method or property, that aspect of the program will fail. Depending on how we write our code, either the failure will be silent (i.e., cause no error message), or it will cause an error message that appears in the Output panel. The error message helps us diagnose the problem.

We certainly strive to use objects appropriately. We don't intentionally call *gotoAndPlay( )* on a *Date* object, because we know that the *gotoAndPlay( )* method isn't supported by the *Date* class. But what happens if we make a typographical error? What if we accidentally invoke *geTime( )* (missing a "t") instead of *getTime( )* on a *Date* object?

```
someDate.geTime( )  // WRONG! No such method!
```

Our call to *geTime( )* will fail because the *Date* class defines no such method.

And what happens if we invoke *indexOf( )* on a value we think is a *String*, but the value turns out to be a *Number*? The call to *indexOf( )* will fail, because the *Number* class doesn't support the *indexOf( )* method. Example 3-1 demonstrates this situation.

Example 3-1. A mistaken datatype assumption

```
// WRONG! This code mistakenly assumes that getDay( ) returns
// a string indicating the day (e.g., "Monday", "Tuesday"),
// but getDay( ) actually returns a number from 0 to 6.
var today;
today = new Date().getDay( );
if (today.indexOf("Friday") == 1) {
  trace("Looking forward to the weekend!");
}

// The correct code should be:
var today;
today = new Date().getDay( );
// Sunday is 0, Monday is 1, ... Friday is 5.
if (today == 5) {
  trace("Looking forward to the weekend!");
}
```

In a large program, these kinds of problems can be exceedingly difficult and time-consuming to track down. In both the *geTime( )* example and the *indexOf( )* example, unless ActionScript reports an error in the Output panel, we'll have a hard time identifying the issue and locating its cause in our program.

To help us recognize and isolate datatype-related problems in our code, we use ActionScript 2.0's *type checking* capabilities. That is, we can ask ActionScript to check the values in our program and warn us with an error message if it detects a value being used in some inappropriate way. But there's a catch: in order to provide this service, ActionScript 2.0 requires that you formally declare the datatype of every variable, property, parameter, and return value that you want checked. To declare the datatype of a variable or property, we use this general form, referred to as *post-colon syntax*:

```
var variableOrPropertyName:datatype
```

Specifying an item's datatype is often called *datatype declaration*. For example, this line of code *declares* that the datatype of the variable `count` is *Number*:

```
var count:Number;
```

We'll learn more about datatype syntax later in this chapter.

As a best practice, in an ActionScript 2.0 program, you should declare the datatype of every variable, property, function parameter, method parameter, function return value, and method return value.

ActionScript 2.0 performs type checking on every variable, property, parameter, and return value that has a declared datatype. If your code attempts to store incompatible types of data in an item that has a declared datatype, a *type error* appears in the Output panel at compile time. Later we'll learn precisely what constitutes an "incompatible type," but for now, you can just assume intuitively that two types are incompatible when they don't match (i.e., *String* and *Number*, *Array* and *Sound*, etc.).

Variables, properties, parameters, and return values *without* a declared datatype are *not* type checked. If you omit the datatype, omit the colon used in post-colon syntax as well.

Type checking helps us guarantee that a program will run the way we intend it to. To see how, let's return to Example 3-1 in which a programmer mistakenly attempted to invoke *indexOf( )* on a numeric value. The source of the programmer's problem was the incorrect assumption that *Date.getDay( )* returns a string, when in fact, it returns a number. The programmer originally assigned the return value of *getDay( )* to the variable `today` without specifying `today`'s datatype:

```
var today;
today = new Date().getDay( );
```

Because the code doesn't specify the datatype of the variable `today`, the ActionScript 2.0 compiler has no way of knowing that the programmer expects `today` to contain a string. The compiler, hence, allows any type of data to be stored in `today`. The preceding code simply stores the return value of *getDay( )* into `today`. Because the return value of *getDay( )* is a number, `today` stores a number, not a string. This eventually leads to a problem with the program.

In ActionScript 2.0, the programmer can prevent the problem from going unnoticed by declaring the intended datatype of the variable `today`, as follows (changes shown in bold):

```
// ":String" is the datatype declaration
var today:String;
today = new Date().getDay( );
```

In this case, the programmer is still "wrong." His assumption that *getDay( )* returns a string is still a problem, but it is no longer a *hidden* problem. Because the programmer has stated his assumption and intent, the ActionScript 2.0 compiler dutifully generates this error:

```
Type mismatch in assignment statement: found Number where String is
required.
```

This error message should elicit great joy. Why? Because *known* errors are usually trivial to fix once you understand the error message. The error message states that the code requires a string but encountered a number instead. We need to work backward to understand the message's meaning. Why did the code "require" a string? It

was just obeying the programmer's request! The compiler thinks the code requires a string because (and for no other reason than) the programmer declared today's datatype as *String*. The error message tells the programmer that he is breaking his own constraints; the programmer declared a string-only data container (the variable today) and tried to place a numeric value (the return value of *getDay( )*) into it.

An inexperienced developer might immediately say, "Aha! The problem is that awful number where a string belongs! I must change the number into a string!" Don't fall into that trap, and don't be misled by the error message.

The programmer originally assumed that *Date.getDay( )* returns a *String* when it in fact returns a *Number*. But the programmer has no control over the value returned by *getDay( )*, which is defined by the *Date* class and not the programmer. So the solution is to accommodate the return value's correct datatype by storing it in a variable of type *Number* instead of type *String*. Example 3-2 demonstrates.

Example 3-2. Fixing a datatype mismatch error

```
// This line declares today's type as a Number.
var today:Number;
// Assign the return value of getDay( ) to today. In this version,
// the variable's datatype matches the datatype of the value returned
// by getDay( ), so no type mismatch error occurs.
today = new Date().getDay( );

// Sunday is 0, Monday is 1, ... Friday is 5.
if (today == 5) {
  trace("Looking forward to the weekend!");
}
```

Example 3-3 demonstrates an alternative case in which the programmer really does need a string for display purposes. As usual, *getDay( )* returns a number, so in this case, the programmer must manually convert the number to a human-readable string. The trick is to use the number returned by *getDay( )* to extract a string from an array of day names.

Example 3-3. One way to derive a string from a number

```
// This line declares today's type as a Number
// and assigns the return value of getDay( ) to today.
var today:Number = new Date().getDay( );

// Populate an array with the names of the days.
var dayNames:Array = ["Sunday", "Monday", "Tuesday", "Wednesday",
                      "Thursday", "Friday", "Saturday"];
// Make a new variable that stores the human-readable day.
var todayName:String = dayNames[today];

// Display the human-readable day in a text field.
currentDay_txt.text = todayName;
// Display the human-readable day in the Output panel.
trace(todayName);
```