downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition

By Colin Moock

Publisher: O'Reilly
Pub Date: December 2002
ISBN: 0-596-00396-X
Pages: 1088

ActionScript for Flash MX: The Definitive Guide is the most complete, up-to-date reference available for the latest version of ActionScript. Author Colin Moock has added hundreds of new code examples to show new Flash MX techniques in the real world. The book's language reference alone has nearly doubled from the first edition, with more than 250 new classes, objects, methods, and properties. You'll find exhaustive coverage of dozens of undocumented, under-documented, and mis-documented features.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Copyright © 2003, 2001 O'Reilly & Associates, Inc.

Printed in the United States of America.

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

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

Nutshell Handbook, the Nutshell Handbook logo, and the O'Reilly logo are registered trademarks of O'Reilly & Associates, Inc. Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and O'Reilly & Associates, Inc. was aware of a trademark claim, the designations have been printed in caps or initial caps. The association between the image of a siren and the topic of ActionScript is a trademark of O'Reilly & Associates, Inc.

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

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Foreword

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

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

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

Flash always has been, and seems destined to remain, the best way to give your web site some pizzazz, but serious web application developers are straining against the limitations of HTML. They are searching for a new platform that offers more attractive, engaging, and usable experiences to their users—a *rich client* — and they are finding Flash to be an ideal delivery vehicle. Flash's cross-platform consistency and ubiquitous distribution base offer a runtime technology upon which developers can build a new breed of web applications that are more interesting and nimble than those that existed previously. I'd wager that you'll be seeing a broad spectrum of new uses for Flash, from multiplayer games to e-commerce to data visualization. And Macromedia is committed to ensuring that Flash keeps up with the new demands placed on it by application developers. ActionScript plays an important role in this new vision for Flash MX. Because the usefulness of the Flash platform depends on the power of its scripting language, we set out to make ActionScript powerful enough to satisfy even the most ambitious web developer.

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

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

Approachable

: Provide an excellent initial experience for new users

Creative

: Enhance Flash's abilities of creative expression

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Power

Beef up ActionScript into a powerful tool for developing complex applications

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

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

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

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

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

—Gary Grossman, Creator of ActionScript, Senior Engineering Manager, Macromedia Flash Team, October 2002.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Preface

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

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

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

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

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Preface

Second Edition Quick Start

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

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

- Chapter 9
- Chapter 10
- Chapter 12
- Chapter 14

See also the revised and new appendixes, especially:

- Appendix C
- Appendix E
- Appendix F
- Appendix G
- Appendix H

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

- *Accessibility* object
- *Button* class
- *Capabilities* object
- *Function* class
- `_global` object

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

- `#initclip` and `#endinitclip` pragmas
- *LoadVars* class
- *LocalConnection* class
- *MovieClip* class (new events and the Drawing API)
- *Object* class
- *setInterval( )* and *clearInterval( )* global functions
- *SharedObject* object
- *Sound* class
- *Stage* object
- *System* object
- *TextField* class
- *TextFormat* class
- Listener Events for *Key*, *Mouse*, *TextField*, and *Stage* (see Table P-1)

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Preface

What's New in Flash MX ActionScript

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

To preview many of the new features in action, visit:

http://www.moock.org/webdesign/lectures/newInMX

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

Table P-1. New features in Flash MX ActionScript

Feature	For details, see...
Drawing API: draw strokes, shapes, and fills at runtime using new *MovieClip* methods	*MovieClip.beginFill( )*, *MovieClip.beginGradientFill( )*, *MovieClip.clear( )*, *MovieClip.curveTo( )*, *MovieClip.endFill( )*, *MovieClip.lineStyle( )*, *MovieClip.lineTo( )*, *MovieClip.moveTo( )*; Section 13.8 in Chapter 13
Load JPEG-format images at runtime	*MovieClip.loadMovie( )*, *loadMovie( )*
Load MP3-format sounds at runtime	*Sound.loadSound( )*
Check the length of a sound and the amount of time it has been playing	*Sound*`.position`, *Sound*`.duration`
Detect when a sound finishes playing	*Sound.onSoundComplete( )*
Create, manipulate, and format text fields at runtime	The *TextField* class, the *TextFormat* class, *MovieClip.createTextField( )*
Mask or unmask a movie clip at runtime	*MovieClip.setMask( )*
Create movie clips from scratch at runtime	*MovieClip.createEmptyMovieClip( )*
Determine a movie clip's depth at runtime	*MovieClip.getDepth( )*
Execute a function or method periodically	*setInterval( )*, *clearInterval( )*

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Manipulate XML, string, and array data faster due to Flash Player performance improvements	The *XML* class, the *String* class, the *Array* class
Store data locally (much like JavaScript cookies)	The *SharedObject* object
Create packaged code modules with *MovieClip* subclasses and components	`#initclip`, `#endinitclip`, *Object.registerClass( )*, *attachMovie( )*; Chapter 14
Communicate between two Flash Players on the same computer	The *LocalConnection* class
Declare global variables	`_global`; Section 5.3 in Chapter 2
Use international characters in the Unicode character set	Section 4.5 in Chapter 4, Appendix C
Define event handlers on movie clips using callback functions	Chapter 10
Use event listeners to respond to events from any object	Chapter 10 and *Key.addListener( )*, *Mouse.addListener( )*, *Stage.addListener( )*, *Selection.addListener( )*, *TextField.addListener( )*
Add button behavior to a movie clip	Section 13.9 in Chapter 13
Control button objects at runtime	The *Button* class
Make content accessible to screen readers for the visually impaired	The *Accessibility* object
Check the movie width and height at runtime, and reposition movie elements when the movie is resized	`Stage.height`, `Stage.width`, *Stage.onResize( )*
Use lexical and nested function scope, or execute a function as a method of an arbitrary object	*Function.call( )*, *Function.apply( )*; Section 2.5.7 in Chapter 2; Section 9.7 in Chapter 9
Access Player and system information such as screen resolution, operating system, and current language	The *Capabilities* object
Capture keyboard and mouse input events with a centralized input API	The *Key* object, the *Mouse* object
Load variables using an intuitive variable loading class rather than the *loadVariables( )* function	The *LoadVars* class
Monitor the download progress of XML or loading variables	*XML.getBytesLoaded( )*, *LoadVars.getBytesLoaded( )*
Control the tab order for buttons, text fields, and movie clips	*`TextField`*`.tabIndex`, *`Button`*`.tabIndex`, *`MovieClip`*`.tabIndex`
Turn off the hand cursor for buttons	*`Button`*`.useHandCursor`, *`MovieClip`*`.useHandCursor`
Add getter/setter properties to an object, and receive notification when a property changes	*Object.addProperty( )*, *Object.watch( )*

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Preface

What's New in the Second Edition

The second edition of *ActionScript for Flash MX: The Definitive Guide* is not merely a "tack-on" update to the first edition (which was titled *ActionScript: The Definitive Guide*). The entire text has been revised and restructured to highlight the latest Flash MX ActionScript features. Nearly every paragraph has been updated, and 400 pages have been added to cover ActionScript's new capabilities. Legacy descriptions of Flash 4 ActionScript syntax have been moved from the body of the book to Appendix C or online technotes. We made this choice to keep the book streamlined, although it is still considerably beefier than the first edition. By the time you read this, Flash Player 6 will be nearly ubiquitous, so it doesn't make sense to cover Flash 4 in detail anymore. We cover enough of it to help you understand and upgrade any legacy code you may own or encounter. We've also paid close attention to changes between Flash 5 and Flash 6 to help you understand the new paradigms and upgrade legacy code. The legacy code examples from the first edition will all remain available at http://www.moock.org/asdg/codedepot.

Updated Code Examples

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

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

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

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

Hundreds of Tweaks

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

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

- `MovieClip._x` discusses *twips* (the minimum distance a clip can be moved).
- `MovieClip._visible` warns that button events don't fire when `_visible` is `false`.
- *XML.parseXML( )* covers CDATA and predefined XML entities (&, <, >, ", and ') at length.
- *MovieClip.getBytesLoaded( )* features a list of possible return values based on the asynchronous execution of *loadMovie( ).*
- Chapter 2 discusses qualified and unqualified variable references and *Hungarian notation.*
- Chapter 4 explicitly contrasts `null` with *delete* and `undefined`.

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

Major Revisions Since the First Edition

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

Chapter 1

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

Chapter 2

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

Chapter 3

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

Chapter 4

- Added coverage of Unicode

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Chapter 5

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

Chapter 6

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

Chapter 8

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

Chapter 9

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

Chapter 10

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

Chapter 11

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

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Chapter 12

- Revised the chapter entirely to focus more squarely on the process of making a class with methods and properties
- Added coverage of Flash MX's *super* keyword, used to invoke a superclass constructor and its methods
- Added a formal discussion of the *prototype chain*
- Added a formal discussion of issues with standard *superclass assignment*
- Added a section on static methods and properties
- Added a description of rendering an object to screen
- Added an object-oriented programming (OOP) application template
- Added an Section 12.9 section
- Added a brief discussion of UML and design patterns

Chapter 13

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

Chapter 14 (all new)

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

Chapter 15 (previously Chapter 14)

- Revised the list of reserved words

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

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

Chapter 16

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

Chapter 17

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

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

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

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

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

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

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

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Preface

What's Not in This Book

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

- Features used exclusively to extend the Flash MX authoring tool (e.g., *CustomActions* and *LivePreview*). These topics are covered in Macromedia's online article "Creating Components in Flash MX" at http://www.macromedia.com/support/flash/applications/creating_comps.

- Macromedia's library of Flash UI Components, which extend the authoring tool beyond the core language. See Appendix G, for a summary of Flash UI Components properties and methods. For resources that cover Flash UI Components in depth, see Section 14.7 in Chapter 14.

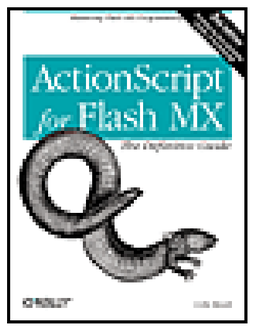
- The Macromedia Flash Communication Server MX (Comm Server) API (e.g., *Remote SharedObject*, *Camera*, *Microphone*, *NetConnection*, and *NetStream*). Comm Server is used to create multiuser web applications with audio and video. See http://www.macromedia.com/software/flashcom/ for details.

- The basics of the Flash MX authoring tool. However, if you are a programmer who is new to Flash, we give you enough hints so you can input the code examples and follow along. To learn Flash MX animation and graphic design, start with the online help and manual; then explore the web sites listed at http://www.moock.org/moockmarks.

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

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir



ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Preface

Undocumented ActionScript Features

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

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

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

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

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

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

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

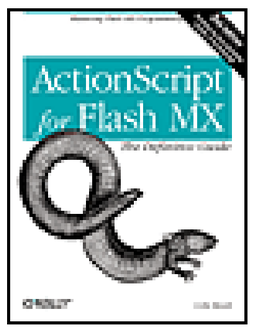
downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

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

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir



ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Preface

Flash Naming Conventions

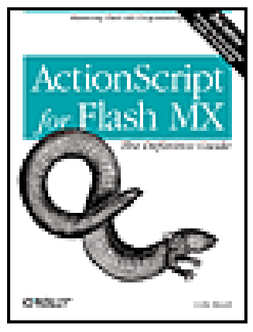
With the introduction of the MX family of products, including Flash MX, Macromedia abandoned a standard numeric versioning system for its Flash authoring tool. The Flash Player, however, is still versioned numerically. Table P-2 describes the naming conventions used in this book for Flash versions.

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

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

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir



ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Preface

What Can ActionScript Do?

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

Timeline Control

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

Interactivity

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

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

Visual and Audio Content Control

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

Programmatic Content Generation

Using ActionScript, we can generate visual and audio content directly from a movie's Library or by duplicating existing content on the Stage. In Flash MX, we can use the *MovieClip* class's Drawing API, *createEmptyMovieClip( )* method, and *createTextField( )* method to create graphics and text from scratch at runtime. Programmatically

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

generated content may serve as a strictly static element — such as a random visual pattern — or as an interactive element — such as a button in a dialog box, an enemy spaceship in a video game, or an option in a pull-down menu.

Server Communication

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

Link to a web page

See *getURL( )*.

Guest book

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

Chat application

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

Multiplayer networked game

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

E-commerce transaction

See the *LoadVars* and *XML* classes.

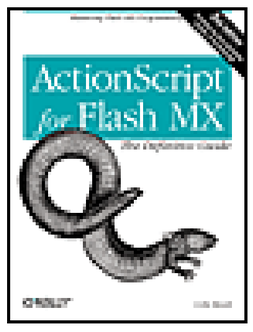
Personalized site involving user registration and login

See the *LoadVars* and *XML* classes.

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

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir



ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Preface

The Code Depot

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

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

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

- A multiple-choice quiz
- A pan-and-zoom image viewer
- Text field tools, such as an array-to-table converter and a configurable text ticker
- An XML-based chat application
- A guest book application
- A custom mouse pointer and button
- An asteroids game code base
- Programmatic motion effects
- Demos of HTML text fields
- Preloaders
- String manipulation
- Interface widgets, such as slider bars and text scrollers
- Mouse trailers and other visual effects
- Volume and sound control

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

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Preface

Showcase

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

Table P-3. ActionScript Showcase

Topic	URL
Experiments in design, interactivity, and scripting	http://www.yugop.com http://www.praystation.com[1] http://www.presstube.com http://www.pitaru.com http://www.flight404.com http://www.bzort-12.com http://www.benchun.net/mx3d/[1] http://www.protocol7.com[1] http://www.uncontrol.com[1] http://flash.onego.ru[1] http://www.figleaf.com/development/flash5[1] http://nuthing.com http://www.deconcept.com http://www.natzke.com
Games	http://www.orisinal.com http://www.gigablast.com http://www.sadisticboxing.com http://www.huihui.de http://www.sarbakan.com http://www.electrotank.com/games/multiuser http://www.titoonic.dk/products/games/spider http://content.uselab.com/acno http://www.neave.com/webgames
Interface, applications, and dynamic content	http://www.mnh.si.edu/africanvoices http://www.curiousmedia.com http://www.smallblueprinter.com http://davinci.figleaf.com/davinci http://host.oddcast.com http://www.enteryourinformation.com/broadmoor/onescreen.cfm

[1] Downloadable *.fla* files provided. Otherwise, only *.swf* files available.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Preface

Typographical Conventions

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

Menu options

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

`Constant width`

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

Italic

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

`Constant width bold`

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

`Constant width italic`

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

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

- Class-level properties are shown with both the class name and property in `constant width`, because they should both be entered verbatim, as shown (e.g., `Stage.width`, `Math.NaN`).
- Instance-level properties are shown with the class or object instance in *`constant width italic`*, because the placeholder should be replaced by a specific instance. The property itself is shown in `constant width` and should be entered as shown (e.g., *`Button`*`.tabEnabled`, where *`Button`* should be replaced with a button instance).
- Method and function names, and the class or object to which they pertain, are always shown in italics and

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

followed by parentheses, as in *MovieClip.duplicateMovieClip( )*. Refer to the ActionScript Language Reference, surrounding material, and nearby examples to determine whether to include the class name literally, as in *TextField.getFontList( )*, or replace it with an instance name, such as *ball_mc.duplicateMovieClip( )*.

- Within the ActionScript Language Reference, for brevity, we often omit the class name when discussing a property or method of the class. For example, when discussing the `htmlText` property of the *TextField* class, when we say "set the `htmlText` property," you should infer from context that we mean, "set the *`someField_txt`*`.htmlText` property, where *`someField_txt`* is the identifier for your particular text field."

- In some cases, an object property contains a reference to a method or callback handler. It wasn't always clear whether we should use `constant width` to indicate that it is a property (albeit one storing a method name) or *italics* and parentheses to indicate it is a method (albeit one stored in a property). If the line between a property referring to a method and the method itself is sometimes blurred, forgive us. To constantly harp on the technical difference would have made the text considerably less accessible and readable.

- When summarizing properties for a class, the properties may be shown in *italics*, rather than `constant width`, to save space. This applies only when the properties are summarized under a *Properties* heading and they aren't followed by parentheses, so it is clear that they're properties and not methods.

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

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

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

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

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Preface

We'd Like to Hear from You

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

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

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

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

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

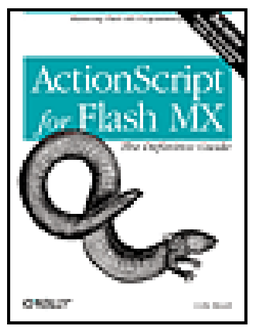
bookquestions@oreilly.com

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

http://www.oreilly.com

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir



ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Preface

Acknowledgments

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

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

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

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

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

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

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

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

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

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

your own.

—Colin MoockToronto, CanadaDecember 2002

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

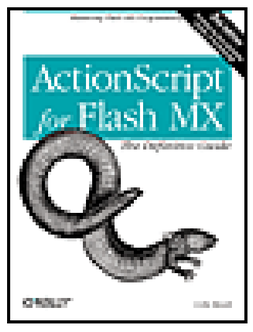
Part I: ActionScript Fundamentals

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

- Chapter 1
- Chapter 2
- Chapter 3
- Chapter 4
- Chapter 5
- Chapter 6
- Chapter 7
- Chapter 8
- Chapter 9
- Chapter 10
- Chapter 11
- Chapter 12
- Chapter 13
- Chapter 14
- Chapter 15
- Chapter 16
- Chapter 17

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir



ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Part I: ActionScript Fundamentals

Chapter 1. A Gentle Introduction for Nonprogrammers

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

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

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

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

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

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

Flash still can't fulfill our request without more information. How big should the ball be? Where should it be placed? In which direction should it begin traveling? How fast should it go? Around which part of the screen should it bounce? For how long? In two dimensions or three? Hmm . . . we weren't expecting all these questions. In reality, Flash doesn't ask us these questions. Instead, when Flash can't understand us, it just doesn't do what we want it to, or it yields an error message. For now, we'll pretend Flash asked us for more explicit instructions, and reformulate our request as a series of steps:

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

Even though we gave our instructions in English, we still had to work through all the logic that governs our bouncing ball in order for Flash to understand us. Obviously, there's more to programming than merely the syntax of

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

programming languages. Just as in English, knowing lots of words doesn't necessarily mean you're a great communicator.

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

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

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

In programming—as in love, politics, and business—effective communication is the key to success. For Flash to understand your ActionScript, you have to get your syntax absolutely correct, down to the last quote, equals sign, and semicolon. And to assure that Flash knows what you're talking about, you must refer only to the world it knows using terms it recognizes. What may be obvious to you is not obvious to a computer. Think of programming a computer like talking to a child: take nothing for granted, be explicit in every detail, and list every step that's necessary to complete a task. But remember that, unlike children, Flash will do precisely what you tell it and nothing that you don't.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Chapter 1. A Gentle Introductionfor Nonprogrammers

1.1 Some Basic Phrases

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

1.1.1 Creating Code

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

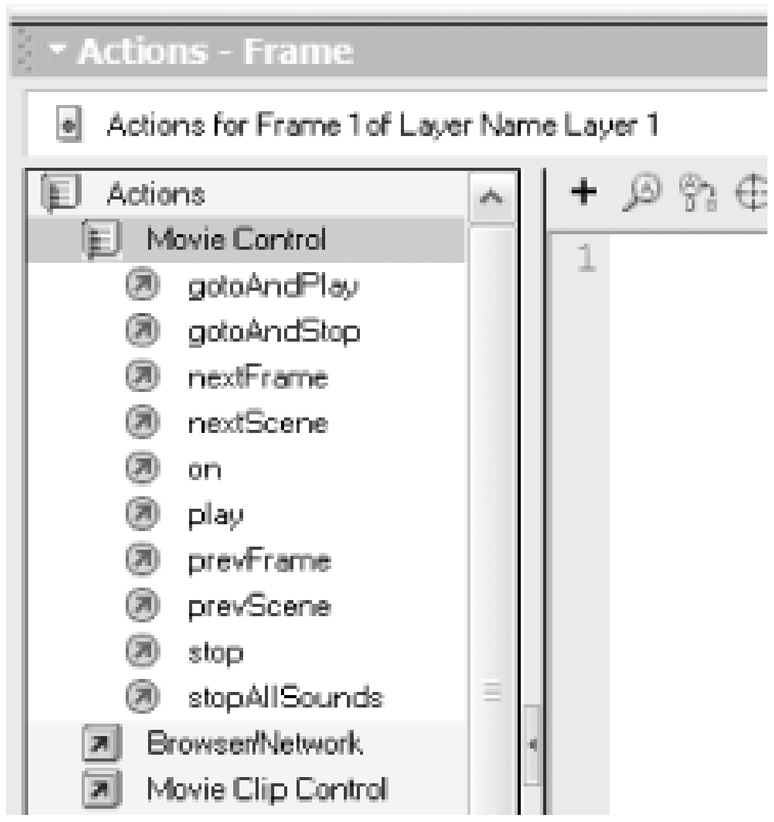
1. Launch Flash with a new blank document.
2. On the main timeline, select frame 1 of layer 1.
3. Select Window Actions (F9). (Note that we use the " " symbol to separate a menu name from a menu option; you should select the "Actions" item from Flash's "Windows" menu.)

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

Figure 1-1. Flash MX Movie Control Actions

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir



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

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

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

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

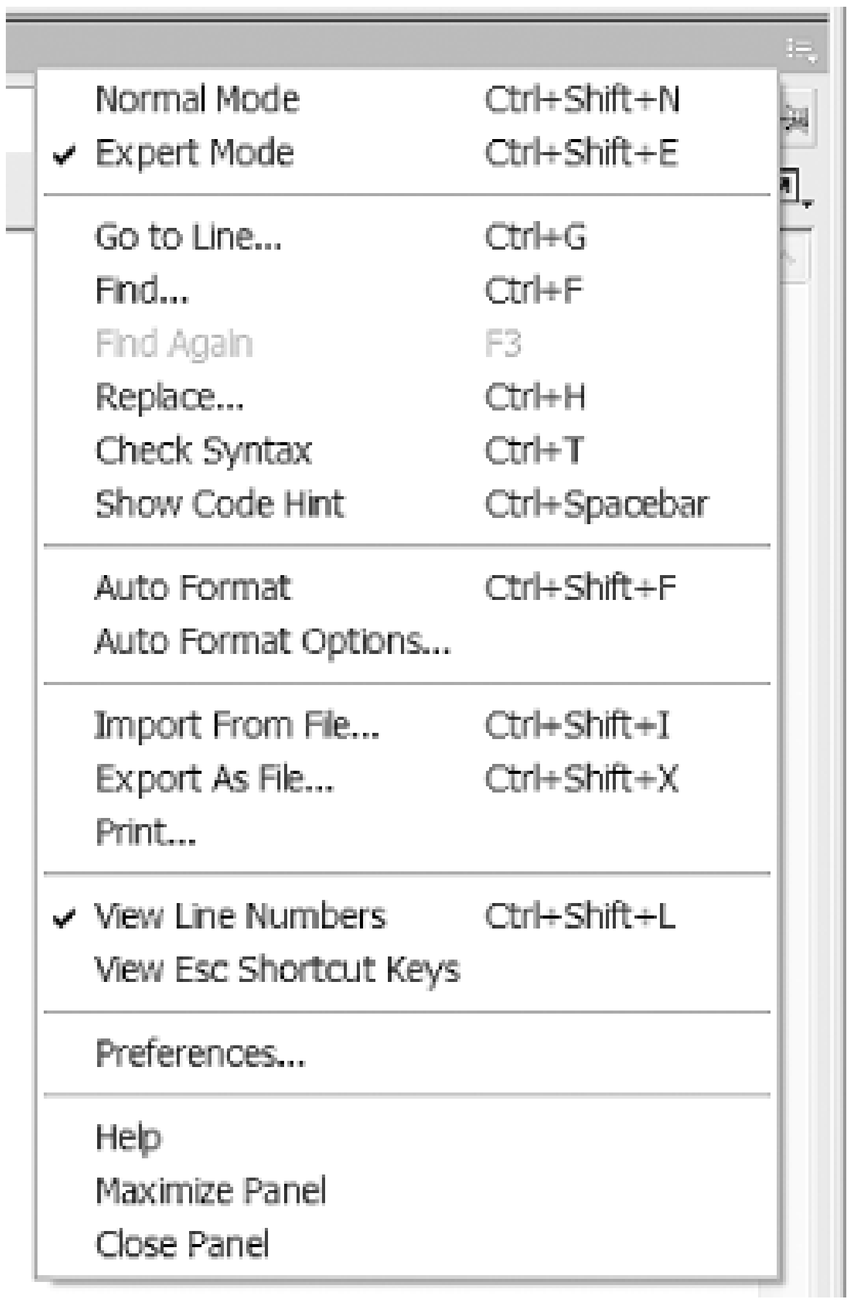
1.1.2 Say Hi to Flash

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

Figure 1-2. Expert Mode selection

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir



For brevity, I'll refer to any pop-up menu in one of Flash's panels as an "Options" menu for the remainder of the book. So I might write, "choose Options Expert Mode in the Actions panel." While you're at it, you can also turn on line numbering using Options View Line Numbers. See also Chapter 16 for more details on the Actions panel.

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

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

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

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

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

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

That line of code constitutes a complete instruction, known as a *statement*. On the line below it, type your second and third lines of code, shown following this paragraph. Replace `your name here` with your first name (whenever

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

you see *`constant-width italicized code`* in this book it means you have to replace that portion of the code with your own content):

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

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

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

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

Figure 1-3. Flash gets friendly

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

1.1.3 Keeping Track of Things (Variables)

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

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

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

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

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

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

1.1.3.1 Official variable nomenclature

It's time for a few formal terms to describe how Flash remembers things. So far, you know that Flash remembers data.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

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

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

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

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

1.1.4 The Wizard Behind the Curtain (the Interpreter)

Recall your first two lines of code:

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

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

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

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

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

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

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

Consider this command as the interpreter would:

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

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

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

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

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

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

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

1.1.5 Extra Info Required (Arguments)

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

```
command(argument);
```

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

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

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

1.1.6 ActionScript's Glue (Operators)

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

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

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

See the + (plus) signs? They're used to join (*concatenate*) our text together and are but one of many available *operators*. The operators of a programming language are akin to conjunctions ("and," "or," "but," etc.) in human languages. They're devices used to combine and manipulate phrases of code. In the *trace( )* example, the plus operator joins the quoted text "Hi there, " to the text contained in the variable `firstName`.

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

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

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

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

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

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

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

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

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

1.1.7 Putting It All Together

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

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

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

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

Line two is pretty much the same as line one:

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

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

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

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

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

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

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Chapter 1. A Gentle Introductionfor Nonprogrammers

1.2 Further ActionScript Concepts

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

1.2.1 Flash Programs

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

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

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

1.2.2 Expressions

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

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

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

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Here we see the variable `message`:

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

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

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

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

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

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

1.2.3 Two Vital Statement Types: Conditionals and Loops

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

1.2.3.1 Making choices using conditionals

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

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

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

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

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

The generic structure of a conditional is:

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

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

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

1.2.3.2 Repeating tasks using loops

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

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

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

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

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

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

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

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

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Here's what our counting example would look like as a *while* loop instead of as a series of individual statements:

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

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

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

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

1.2.4 Modular Code (Functions)

So far, your longest script has consisted of five lines of code. But it won't be long before those five lines become 500, or maybe even 5,000. When you want to manage your code more efficiently, reduce your work, and make your code easier to apply to multiple scenarios, you'll learn to love *functions*. A function is a packaged series of statements that performs some useful action or calculation. In practice, functions serve mostly as reusable blocks of code.

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

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

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

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

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

```
var area5 = height5 * width5;
```

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

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

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

```
return height * width;
```

After we create a function, we may run the code it contains from anywhere in our movie by using the function's name. In our example we called the *area( )* function five times, passing it the `height` and `width` values it expects each time: *area(10, 15)*, *area(11, 16)*, and so on. The result of each calculation is returned to us, and we store those results in the variables `area1` through `area5`. This is nice and neat, and much less work than the nonfunction version of our code shown earlier. Enclosing reusable code within functions save us time and reduces the number of lines of code needed. It also makes our code much more legible and modular. Someone reading the code can probably guess what the *area( )* function does, based solely on its name.

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

1.2.4.1 Built-in functions

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

1.2.5 Objects and Object-Oriented Programming

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

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

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

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

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

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

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

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

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

1.2.6 Movie Clip Instances

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

Every movie clip has a symbol definition that resides in the Library of a Flash movie. We can add many copies, or *instances*, of a single movie clip symbol to a Flash movie by dragging the clip from the Library onto the Stage. We can also create movie clips at runtime via ActionScript. All visual programming in Flash involves some degree of movie clip instance control. For example, a bouncing ball is nothing more than a movie clip instance that is repositioned on the Stage repetitively. Using ActionScript, we can alter an instance's location, size, current frame, rotation, and so forth, during the playback of our movie.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

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

1.2.7 The Event-Based Execution Model

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

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

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

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

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

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

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

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

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

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

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

A function that is executed when an event occurs is known as a *callback function*. As we'll learn in Chapter 10, within a callback function, the keyword `this` refers to the object that defined the event handler (in our case, `rotateButton`). In the case of a button reacting to a mouseclick, `this` refers to the button that was clicked. Using object-oriented syntax, the movie clip in which the button resides is referred to as `this._parent` (the

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

movie clip is the button's `_parent` because it contains the button). Finally, we set the rotation of the parent movie clip to 45 degrees by assigning 45 to `this._parent._rotation`:

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

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

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

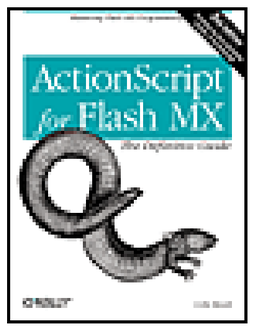
```
rotateButton.onRelease = function ( ) {
  this._parent._rotation = 45;
};
```

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

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

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir



ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Chapter 1. A Gentle Introductionfor Nonprogrammers

1.3 Building a Multiple-Choice Quiz

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

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

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

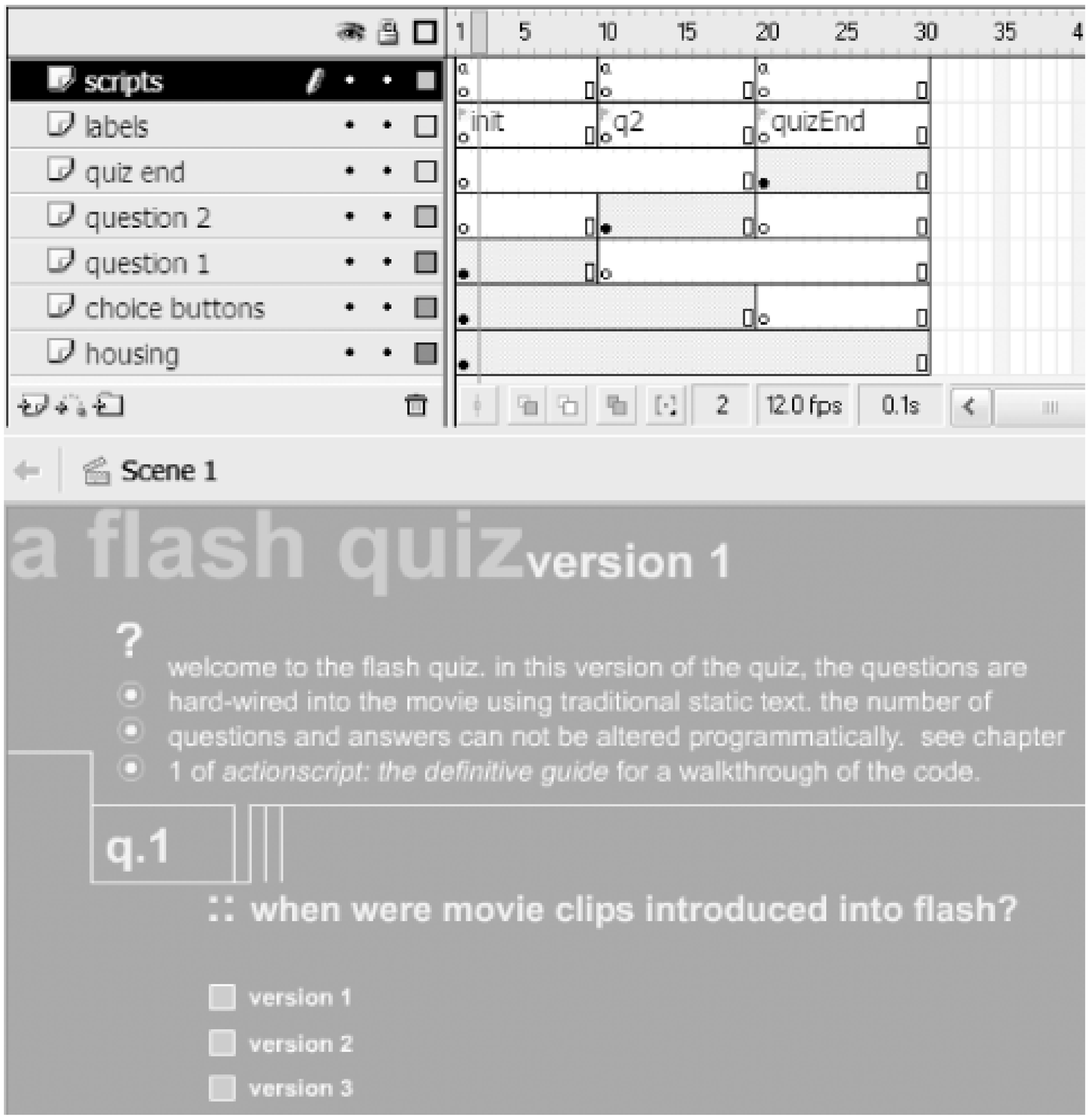
1.3.1 Quiz Overview

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

Figure 1-4. A Flash quiz

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir



1.3.2 Building the Layer Structure

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

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

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

scripts
labels
quiz end
question 2
question 1
choice buttons

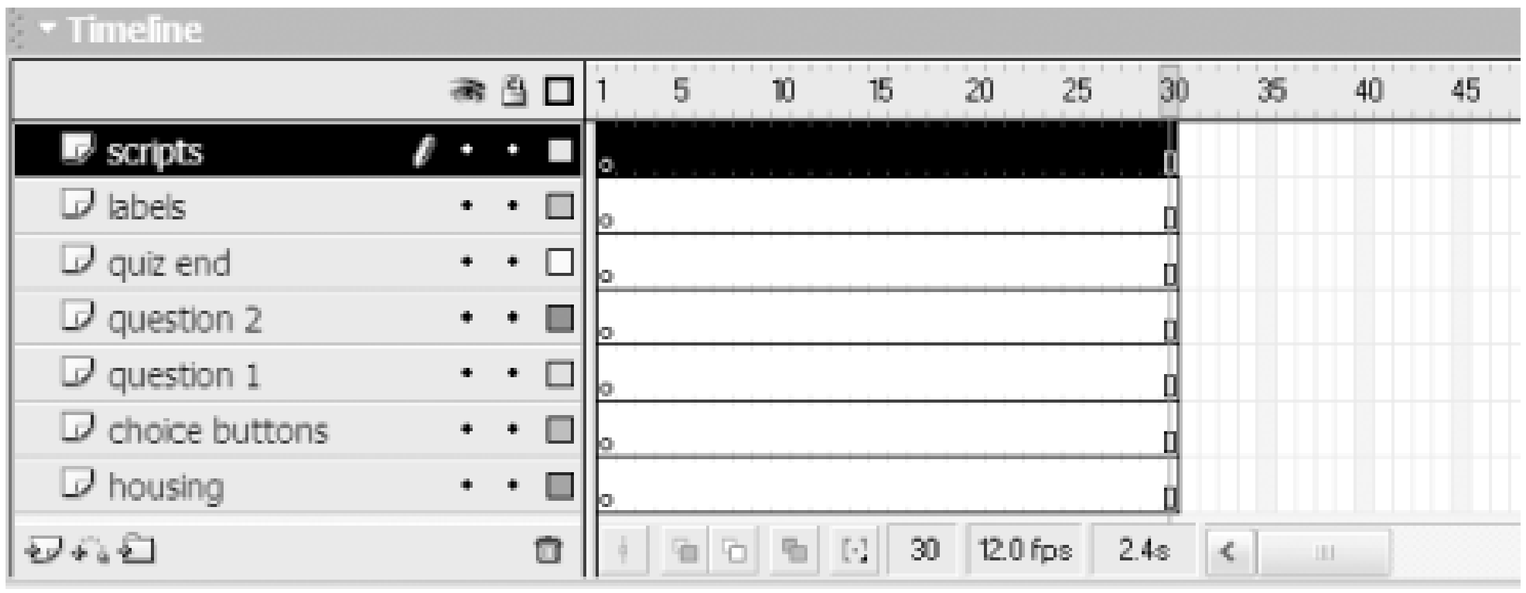
downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

housing

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

Figure 1-5. Quiz timeline initial setup



1.3.3 Creating the Interface and Questions

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

Follow these steps to create the questions and quiz title:

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

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

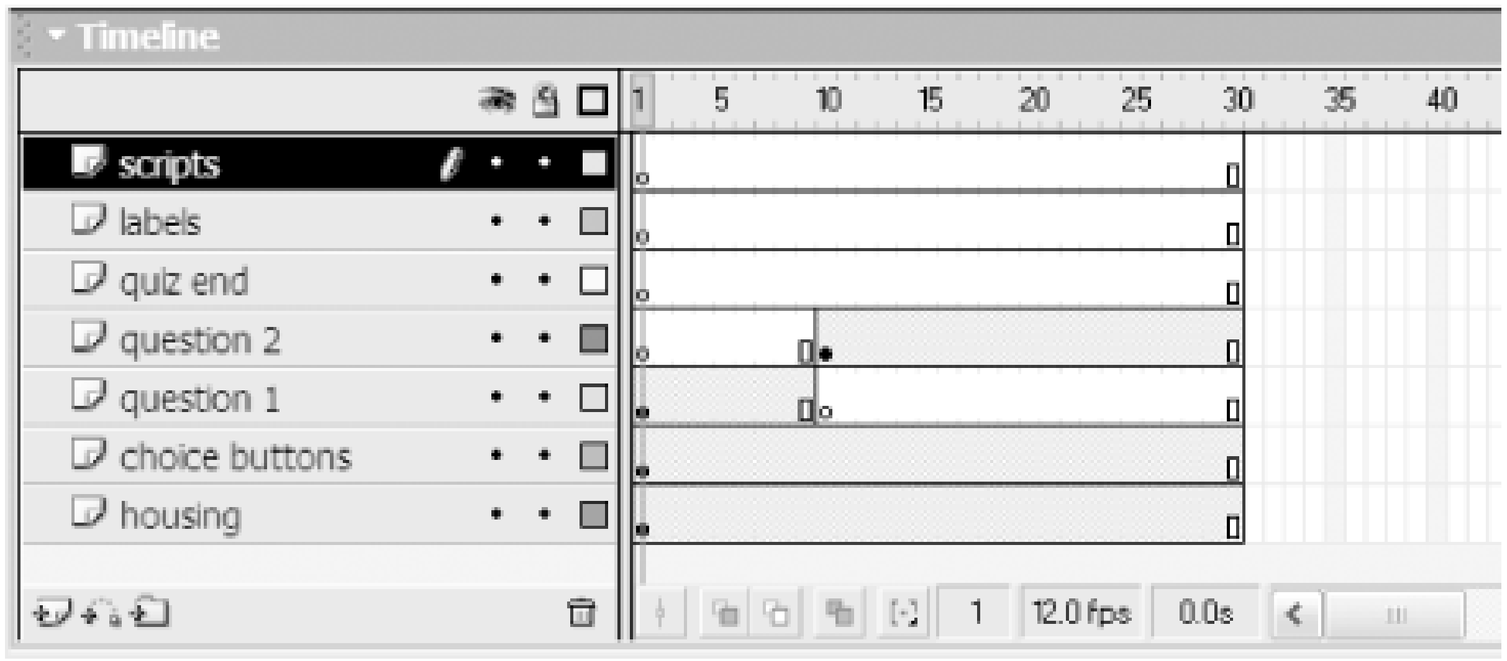
1. Create a simple button symbol (Insert New Symbol Button) that looks like a checkbox or radio button and measures no higher than a line of text (see the buttons in Figure 1-4).
2. At frame 1 of the *choice buttons* layer, next to your three Question 1 answers, place three instances of your checkbox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

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

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

Figure 1-6. Quiz timeline with two questions



1.3.4 Initializing the Quiz

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

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

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

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

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

Example 1-1. Init code for quiz

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

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

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

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

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

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

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

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

```
var favoriteColor;
```

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

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

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

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

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

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

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

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

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

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

1. Select frame 1 of the *scripts* layer.
2. Use Window Actions (F9) to open the Actions panel.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

3. Make sure you're using Expert Mode, which can be set as a permanent preference via the pop-up Options menu in the top right corner of the Actions Panel.
4. In the right side of the Actions panel, type the *init* code as shown earlier in Example 1-1.

Variable Naming Styles

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

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

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

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

1.3.5 Adding Frame Labels

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

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

```
gotoAndStop(10);
```

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

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

points on the timeline.

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

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

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

1.3.6 Scripting the Answer Buttons

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

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

Figure 1-7. The answer buttons

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

Example 1-2. Question 1 button code

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

// Code executed when button 2 is pressed.
choice2_btn.onRelease = function ( ) {
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

```
  this._parent.q1answer = 2;
  this._parent.gotoAndStop("q2");
};

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

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

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

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

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

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

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

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

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

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

The first line sets our variable `q1answer` to 1 (the other answer buttons set it to 2 or 3). The `q1answer` variable stores the user's answer for the first question. However, as a matter of good form, we provide not just the name of the `q1answer` variable, but also its location in relation to the button. The expression `this._parent` means "this button's parent movie clip" or, synonymously, "the movie clip that contains this button." The keyword `this` represents the button itself, while `_parent` is the button's parent movie clip. So, the complete statement

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

`this._parent.q1answer = 1` means, "In this button's parent movie clip, set the variable `q1answer` to 1." If you're new to this syntax, you may find it overly verbose. However, as we'll learn in Chapter 2, every variable has a home (usually a movie clip), and we're normally required to provide not only a variable's name but also its location (e.g., `this._parent.q1answer`, as opposed to just `q1answer`).

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

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

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

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

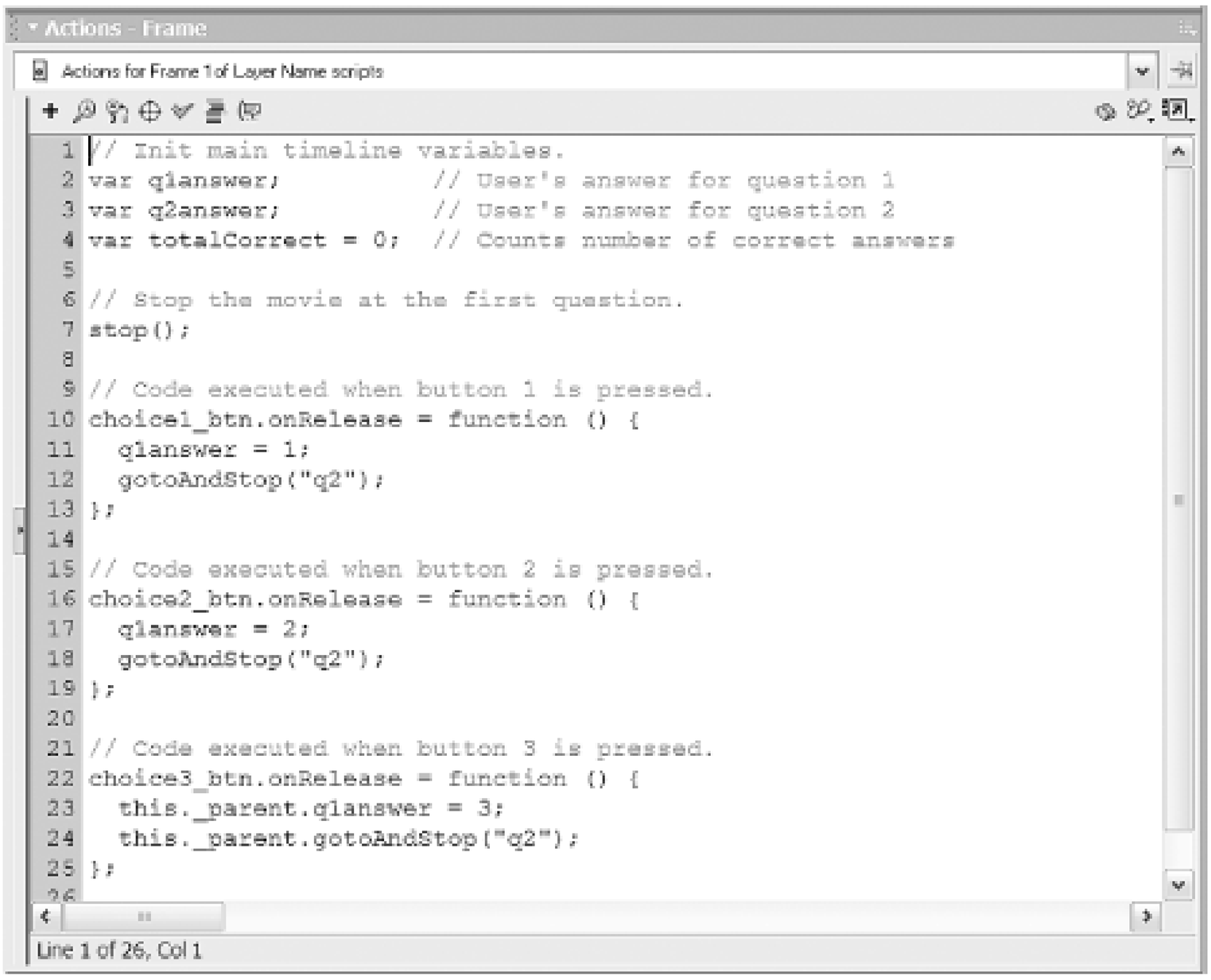
- Each specifies a different button instance name (`choice1_btn`, `choice2_btn`, `choice3_btn`).
- Each sets a different value for `q1answer`, recording the user's actual answer to the question.

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

Figure 1-8. Code on frame 1

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir



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

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

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

Example 1-3. Question 2 button code

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

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

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

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

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

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

1.3.7 Building the Quiz End

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

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

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

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

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

Example 1-4. Quiz end code

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

// Create an onscreen text field do display the user's score.
```

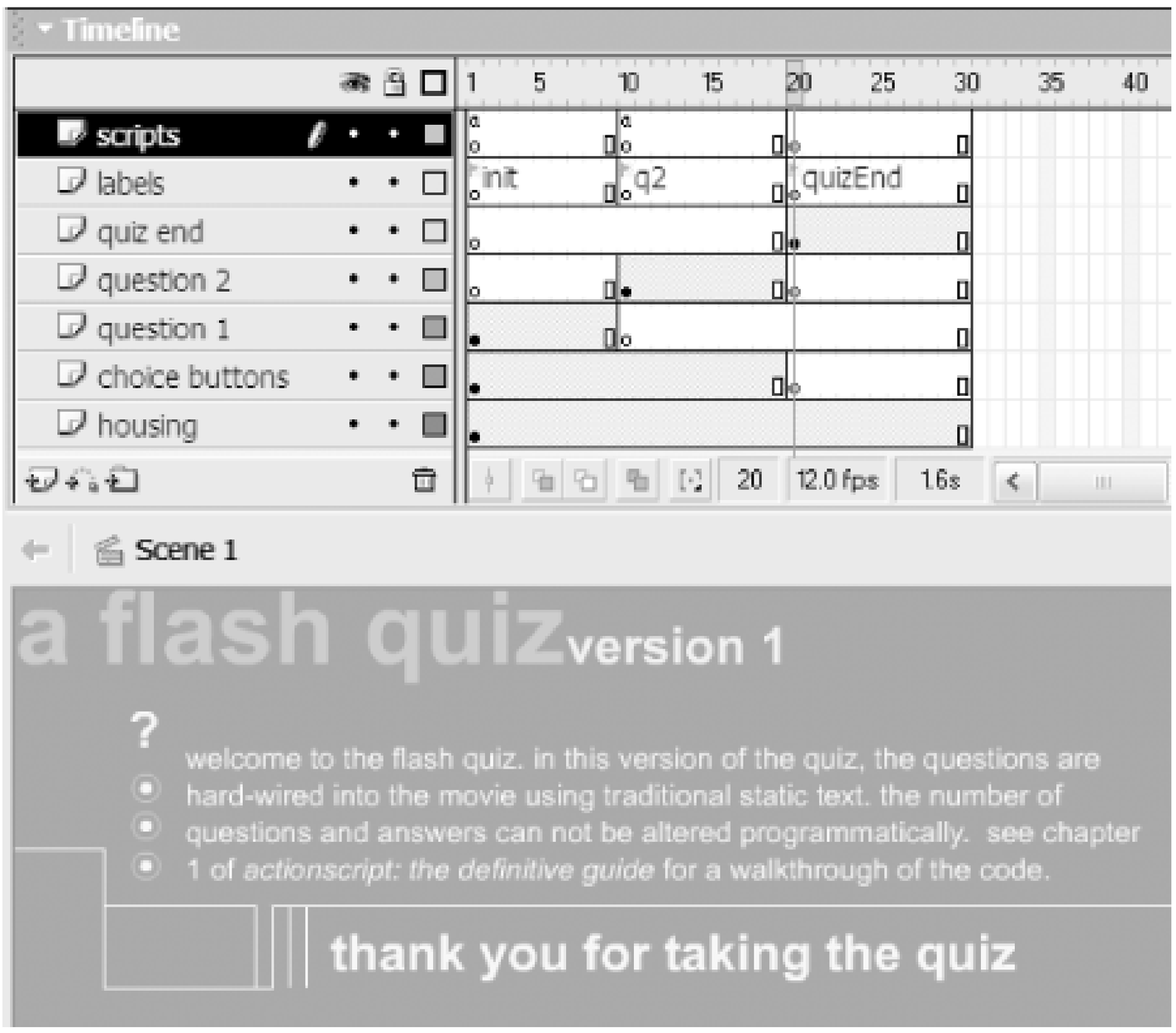
downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

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

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

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

Figure 1-9. Judgment day




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

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

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

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

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

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

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

So, instead of using this code:

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

we normally write:

```
totalCorrect++;
```

which does exactly the same thing, but more succinctly.

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

```
}
```

Lines 5 through 7 are another condition:

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

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

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

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

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

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

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

1.3.8 Testing Our Quiz

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

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

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

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

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

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Chapter 1. A Gentle Introductionfor Nonprogrammers

1.4 Onward!

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

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

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

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

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Part I: ActionScript Fundamentals

Chapter 2. Variables

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

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

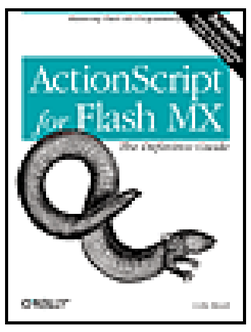
Once a variable is created, we can put new data into it as often as we want—much like depositing money into an account. Or, we can find out what's in a variable by using the variable's name—much like checking an account's balance. If we no longer need our variable, we can "close the account" by deleting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

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

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

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir



ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Chapter 2. Variables

2.1 Creating Variables (Declaration)

Creating a variable is called *declaration* . Declaration is the "open an account" step of our bank metaphor, where we formally bring the variable into existence. When a variable is first declared, it is empty—a blank slate waiting to be written upon. In this state, a variable contains a special value called `undefined` (indicating the absence of data).

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

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

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

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

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

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

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

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

2.1.1 Automatic Variable Creation

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

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

2.1.2 Legal Variable Names

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

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

- Must be composed exclusively of letters, numbers, dollar signs ($) and underscores (No spaces, hyphens, or other punctuation marks are allowed.)
- Must start with a letter, an underscore (e.g., `_someVar`), or a dollar sign (e.g., `$someVar`)
- Must not exceed 255 characters (Okay, okay, that's a lie, but reevaluate your naming scheme if your variable names exceed 255 characters.)
- Are case-insensitive (Upper- and lowercase letters are treated identically, but you should be consistent nonetheless.)

These are legal variable names:

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

These are illegal variable names that cause errors:

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

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

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

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

Table 2-1. Flash MX code hinting suffixes

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Suffix	Datatype represented
_mc	MovieClip
_array	Array
_str	String
_btn	Button
_txt	TextField
_fmt	TextFormat
_date	Date
_sound	Sound
_xml	XML
_xmlsocket	XMLSocket
_color	Color
_video	Video
_ch	FCheckBox[1]
_pb	FPushButton[1]
_rb	FRadioButton[1]
_lb	FListBox[1]
_sb	FScrollBar[1]
_cb	FComboBox[1]
_sp	FScrollPane[1]

[1] Flash UI Components (see Appendix G)

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

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

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

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

2.1.2.1 Creating dynamically named variables

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

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

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

2.1.3 Declare Variables at the Outset

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

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

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

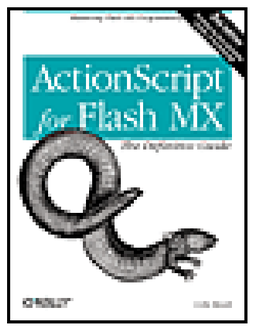
We can give variables an initial value at the same time we create them, as follows:

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

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

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir



ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Chapter 2. Variables

2.2 Assigning Values to Variables

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

```
variableName = value;
```

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

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

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

Here are two more variable assignment examples:

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

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

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

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

When the statement *`y = 1 + 5;`* is executed, 1 is first added to 5, yielding 6, and then 6 is assigned to `y`. The expression on the right side of the equals sign is *evaluated* (calculated or resolved) before assigning the result to the variable on the left side. Here, we assign an expression that contains the variable `y` to another variable, `z`:

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

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

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

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

statement:

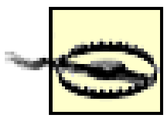
```
myList = ["John", "Joyce", "Sharon", "Rick", "Megan"];
```

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

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

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

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



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

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

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

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

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

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

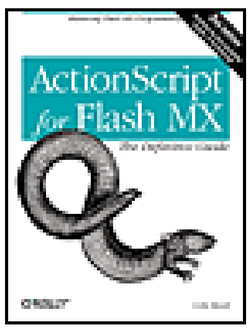
Most likely, the programmer intended to compare the current value of `x` to 5:

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

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

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir



ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Chapter 2. Variables

2.3 Changing and Retrieving Variable Values

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

Example 2-1. Changing variable values

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

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

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

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

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

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

2.3.1 Checking Whether a Variable Has a Value

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

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

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

```
}
```

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

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

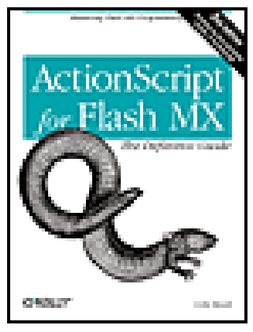
Both the *typeof* and the strict inequality techniques just shown prevent Flash from performing automatic datatype conversion when checking if *`someVariable`* is `undefined` . By contrast, due to automatic datatype conversion, the regular inequality operator (`!=`) checks not only whether *`someVariable`*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

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

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

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir



ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Chapter 2. Variables

2.4 Types of Values

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

2.4.1 Automatic Typing

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

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

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

In languages that do not perform automatic retyping, such as C++ and Java, data of the wrong type is converted to the variable's existing datatype (or it causes an error if conversion cannot be performed). By comparison, VB.NET allows the programmer to decide whether the compiler should enforce strict typing. Strict typing may seem cumbersome, but it can prevent the errors that unsuspecting programmers may encounter due to automatic and dynamic typing, which we'll consider in the following sections.

2.4.2 Automatic Value Conversion

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

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

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

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

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

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

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

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

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

2.4.3 Determining the Type Manually

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

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

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

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

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

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Chapter 2. Variables

2.5 Variable Scope

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

2.5.1 Timeline Variables

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

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

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

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

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

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

```
trace(x);
```

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

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

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

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

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

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

and on frame 15 we add:

downloaded from: lib.ommolketab.ir

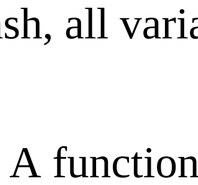
downloaded from: lib.ommolketab.ir

```
trace(password);
```

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

```
undefined
let_me_in
```

Here's why: when the movie's playhead reaches frame 15, Flash displays the value of `password` in the Output window. Th first time through, `password` does not exist, so Flash displays `undefined` . However, when the playhead reaches frame 30, `password` is created and assigned the value "let_me_in". Then the playhead is sent back to frame 15 (*gotoAndStop(1*), causing frame 15's code to execute again. This time, even though `password` is defined on a later frame than our *trace(password)* statement, it is still part of the same timeline; so, its value exists, and Flash displays it in the Output window.



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

2.5.2 Variable Accessibility (Scope)

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

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

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

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

2.5.3 Movie Clip Variables and Global Variables

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

2.5.3.1 Scenario 3: Variables declared on separate timelines

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

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

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

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

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

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

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

First question: if we attach the following code:

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

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

Answer:

```
undefined
undefined
```

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

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

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

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

```
3
undefined
```

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

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

We refer to variables attached to timelines as *timeline variables* or *movie clip variables* . Now let's create a *global variable* tha

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

is directly accessible to all the scripts in a movie (any timeline). We'll put the following code on frame 1 of the `square` timeline, even though the code would have the same effect from any movie clip:

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

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

```
trace(day);
```

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

```
Monday
```

As we'll see in Section 2.5.7, later in this chapter, when a timeline variable and a global variable both have the same name, the timeline variable takes precedence. For example, if we create a timeline variable, `day` , inside our `circle` movie clip, as follows:

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

and then we check the value of `day` inside `circle` :

```
trace(day);
```

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

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

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

For example:

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

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

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

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

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

We discuss this technique (and the reason it works) in Section 12.5.3.5, in Chapter 12. In Flash MX, the `_global` object is the preferred way to store global variables.

2.5.4 Accessing Variables on Different Timelines

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

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

```
movieClipInstanceName.variableName
```

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

```
square.x
```

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

```
circle.y
```

A reference to a variable that includes the variable's location is known as a *qualified reference* . Qualified references tell Flash where to find the variable we're referring to; unqualified references require Flash to make assumptions about where the variable resides, as discussed later in Section 2.5.7. Hence, it's always good form to use qualified references to variables.

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

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

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

2.5.4.1 The _root and _ parent properties

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

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

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

```
_root.square.x
```

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

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

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

There are times, however, when we want to refer to variables on other timelines without referring to the main timeline of a movie. To do so, we use the `_parent` property, which refers to the timeline upon which the current movie clip instance

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

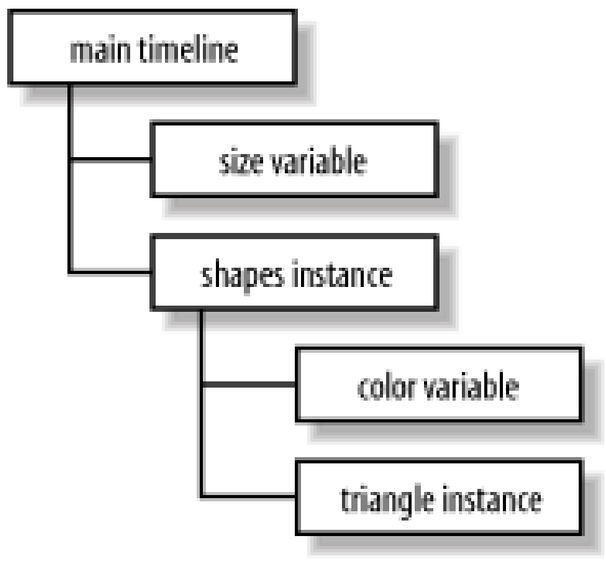
resides. For example, from code attached to a frame of the clip `square` , we can refer to variables on the timeline that contains `square` using this syntax:

```
_parent.myVariable
```

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

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

Figure 2-1. A sample movie clip hierarchy



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

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

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

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

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

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

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

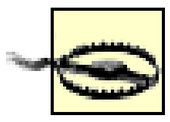
Using the `_parent` property twice in succession ascends two levels up the movie clip hierarchy, which in this context brin us to the main timeline of the movie.

Your approach to variable addressing will depend on what you want to happen when you place instances of a movie clip symbol on various timelines. In our `triangle` example, if we wanted our reference to `color` to always point to `color` defined in the `shapes` clip, we would use the `_root` syntax, which gives us a fixed reference to `color` in `shapes` . Bu if we wanted our reference to `color` to refer to a different `color` variable, depending on which timeline held a given

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

`triangle` instance, we would use the `_parent` syntax.



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

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

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

For example:

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

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

```
myPicturesMainTimeline.variableName
```

instead of:

```
_root.variableName
```

2.5.4.2 Accessing variables on different document levels

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

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

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

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

2.5.5 Movie Clip Variable Life Span

Earlier, we said that the scope of a variable answers two questions: (a) how long does the variable exist? and (b) from where i

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

our code can we set or retrieve the variable's value? For movie clip variables, we have shown the factors involved in answerir the second question. But we skipped answering the first question. Let's return to it now with one final variable-coding scenari

2.5.5.1 Scenario 4: Life span of movie clip variables

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

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

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

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

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

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

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

2.5.6 Local Variables

Movie clip variables are scoped to movie clips and persist as long as the movie clip on which they are defined exists. Sometimes, that's longer than we need them to live. For situations in which we need a variable only temporarily, ActionScript offers variables with *local* scope (i.e., *local variables*), which live for a much shorter time than normal movie clip variables.

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

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

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

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

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

therefore, like `counter` to die after the *displayElements( )* function has finished.

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

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

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

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

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

The effect of the statement `height = 10;` depends on whether `height` is a local variable or movie clip variable. If `height` is a previously declared local variable (which it is not in the example at hand), the statement `height = 10;` simply modifies the local variable's value. If there is no local variable named `height` , as in this case, the interpreter creates movie clip (nonlocal) variable named `height` and sets its value to 10. As a nonlocal variable, `height` persists even after the function finishes.

Example 2-4 demonstrates local and nonlocal variable usage.

Example 2-4. Local and nonlocal variables

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

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

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

```
var myColor = "blue";
function hexRed () {
   var myColor = "#FF0000";
   return myColor;
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

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

2.5.7 The Scope Chain

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

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

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

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

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

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

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

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

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

Note that, technically, each scope chain is implemented internally as a series of objects used by the interpreter to look up variables. See Chapter 9 of this book and David Flanagan's eloquent description of JavaScript's variable scope in *JavaScript:*

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

The Definitive Guide (O'Reilly & Associates, Inc.). Bear in mind, however, that ActionScript adds movie clips to JavaScript's global and local execution contexts.

2.5.8 Event Handler Scope

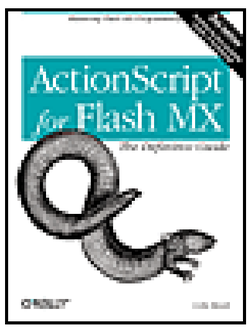
As you'll see in Chapter 10, different kinds of event handlers have different scopes in Flash. Table 2-2 provides a summary of event handler scope. For each type of handler, an unqualified, nonlocal variable declaration (one without a *var* statement) will create a new variable in the movie clip listed in the *Scope chain* column of the table. We'll examine this topic more closely in Chapter 10.

Table 2-2. Event handler scope

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

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir



ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Chapter 2. Variables

2.6 Loading External Variables

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

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

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

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

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

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

For complete details, see:

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

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

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

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

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

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

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

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

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

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Chapter 2. Variables

2.7 Some Applied Examples

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

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

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

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

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

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

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

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

downloaded from: lib.ommolketab.ir

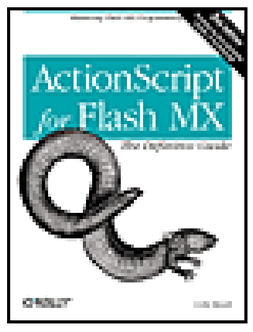
downloaded from: lib.ommolketab.ir

Example 2-8. A Fahrenheit/Celsius temperature converter

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

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

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

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir



ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

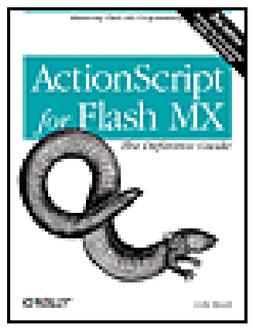
Chapter 2. Variables

2.8 Onward!

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

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir



ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Part I: ActionScript Fundamentals

Chapter 3. Data and Datatypes

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

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Chapter 3. Data and Datatypes

3.1 Data Versus Information

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

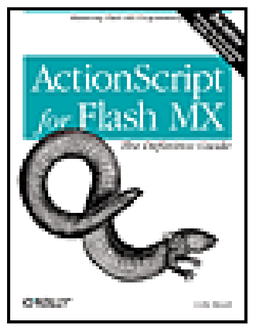
```
01010101010101011010101110101010101010000010101010101101010101
10101010101010101110101010101010101010101010101111101010101010101
01010101010101010101010101110101010101010101010101010101010101010
```

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

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

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir



ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Chapter 3. Data and Datatypes

3.2 Retaining Meaning with Datatypes

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

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

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

3.2.1 The ActionScript Datatypes

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

string

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

number

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

boolean

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

null and undefined

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

array

: For lists of one or more pieces of data.

movieclip

: For manipulating movie clip instances.

object

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

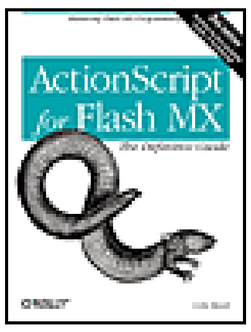
downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

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

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir



ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Chapter 3. Data and Datatypes

3.3 Creating and Categorizing Data

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

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

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

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

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

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

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

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

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

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

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

3.3.1 Automatic Literal Typing

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

```
"animal"                // Quotation marks identify "animal" as a string
1.35                    // If it contains only integers and a decimal point,
                        // it is a number
true                    // Special keyword true identifies this as a Boolean
null                    // Special keyword null identifies this as the null type
undefined               // Special keyword undefined identifies the undefined ty
["hello", 2, true]      // Square brackets and values separated by commas
                        // indicate that this is an array
{x: 234, y: 456}        // Curly braces and property name/value pairs separated
                        // by commas indicate that this is an object
```

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

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

3.3.2 Automatic Complex Expression Typing

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

```
pointerX = _xmouse;
```

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

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

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

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

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

Combining disparate types of data or using a datum in a context that does not match the expected datatype causes ambiguity. This forces the interpreter to perform an automatic datatype *conversion* according to arbitrary, but predictable, rules. Let's examine the cases in which automatic conversions will occur and the results of converting a datum from one type to another.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Chapter 3. Data and Datatypes

3.4 Datatype Conversion

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

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

3.4.1 Automatic Type Conversion

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

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

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

```
999 - NaN;
```

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

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

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

is interpreted as:

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

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

To answer the first and second questions, we need to consult the appropriate topics elsewhere in this book (e.g., to determine

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

datatype is expected in a conditional statement, see Chapter 7).

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

Table 3-1. Converting to a number

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

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

Table 3-2. Converting to a string

Original data	Result after conversion
`undefined`	"" (the empty string)
`null`	"`null` "
Boolean	"`true` " if the original value was `true` ; "`false` " if the original value was `false` .
`NaN`	"`NaN` "
`0`	"`0` "
`Infinity`	"`Infinity` "
`-Infinity`	"`-Infinity` "
Other numeric value	String equivalent of the number. For example, `944.345` becomes "`944.345` ".
Array	A comma-separated list of element values.
Object	The value that results from calling *toString( )* on the object. By default, the *toString ( )* method of an object "`[object Object]` ". The *toString( )* method can be customized to return a more useful result (e.g., *to* a *Date* object returns: "`Sun May 14 11:38:10 EDT 2000` ").
Movieclip	The path to the movie clip instance, given in absolute terms starting with the document level in the Player. I example, "`_level0.ball` ".

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

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

Table 3-3. Converting to a Boolean

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

3.4.2 Explicit Type Conversion

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

3.4.2.1 Converting to a string with the toString( ) method

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

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

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

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

3.4.2.2 Converting to a string with the String( ) function

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

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

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

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

3.4.2.3 Converting to a string with empty string concatenation

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

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

3.4.2.4 Converting to a number with the Number( ) function

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

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

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

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

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

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

3.4.2.5 Converting to a number by subtracting zero

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

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

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

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

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Here are some *parseInt( )* examples:

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

// Convert decimal to hexadecimal.
(255).toString(16);             // Yields: ff
// Convert hexadecimal to decimal.
parseInt("0xFF");               // Yields 255
```

The *parseFloat( )* function returns the first floating-point number that appears in a string, provided that the string's first nonbla character is a valid numeric character. (A floating-point number is a positive or negative number that contains a decimal value 10.5 or 345.678.) Like *parseInt( )* , *parseFloat( )* yields the special numeric value `NaN` if the string's first nonblank character valid numeric character. The number extracted by *parseFloat( )* is the numeric conversion of the series of characters that starts first nonblank character in the string and ends with the character before the first nonnumeric character (any character other tha 9, a decimal point, or an *e* or *E* when used for exponential notation).

Here are some *parseFloat( )* examples:

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

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

3.4.2.7 Converting to a Boolean

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

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

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

3.4.3 Conversion Duration

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

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

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

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

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

3.4.4 Determining the Type of an Existing Datum

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

```
typeof expression;
```

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

Table 3-4. Return values of typeof

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

Here are a few examples:

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

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

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

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

// Check all the properties of the main timeline.
for (prop in _root) {
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

```
  // If the current property is a movie clip...
  if (typeof _root[prop] =  = "movieclip") {
    // ...add it to the clips array.
    childClips.push(_root[prop]);
  }
}

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

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Chapter 3. Data and Datatypes

3.5 Primitive Data Versus Composite Data

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

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

Primitive datatypes are, as their name suggests, simple. They can hold text messages, frame numbers, secret passwords, and so on, but they don't readily accommodate higher levels of complexity. For more elaborate data handling—such as simulating the physics of a dozen bouncing balls or managing a quiz with 500 questions and answers—we turn to *composite* datatypes. Using composite data, we can manage multiple pieces of related data as a single datum.

ActionScript supports the following composite datatypes: *array*, *object*, and *movieclip*. Technically, functions are a type of object and are therefore considered composite data, but we rarely manipulate them as such. See Chapter 9 for more about functions as a datatype.

Whereas a single number is a primitive datum, a list (i.e., an *array*) of multiple numbers is a composite datum. Here's a practical example of how composite datatypes are useful: suppose we want to store the name of a customer named Derek. We can create a variable that stores Derek's name as a primitive value, like this:

```
var custName = "Derek";
```

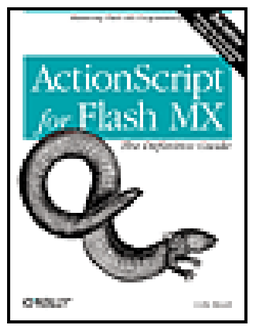
However, this approach gets pretty cumbersome once we add more customers. We're forced to use sequentially named variables to keep track of our customers—`cust1Name`, `cust2Name`, `cust3Name`, and so on. Yuck! But if we use an array, we can store our information much more efficiently:

```
customers = ["Derek", "James", "Joe"];
```

Now that's nice and tidy. We'll learn much more about composite datatypes in the coming chapters.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir



ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Chapter 3. Data and Datatypes

3.6 Copying, Comparing, and Passing Data

There are three fundamental ways to manipulate data; we can *copy* it (e.g., assign the value of variable `x` to variable `y`), *compare* it (e.g., check whether `x` equals `y`), and *pass* it (e.g., supply a variable to a function as an argument). Primitive data values are copied, compared, and passed quite differently than composite data. When primitive data is copied to a variable, that variable gets its own unique and private copy of the data, stored separately in memory. Hence, the following lines of code cause the string "Dave" to be stored twice in memory, once in the memory location reserved for `name1` and again in the location reserved for `name2`:

```
name1 = "Dave";
name2 = name1;
```

We say that primitive data is copied *by value* because the data's literal value is stored in the memory location allotted to the variable. In contrast, when composite data is copied to a variable, only a *reference* to the data (and not the actual data) is stored in the variable's memory slot. That reference tells the interpreter where the actual data is kept (i.e., its address in memory). When a variable that contains composite data is copied to another variable, it is the *reference* (often called a *pointer*) and not the data itself that is copied. Hence, composite data is said to be copied *by reference*.

This makes good design sense, because it would be grossly inefficient to duplicate large arrays and other composite datatypes, but it has important consequences for our code. When multiple variables are assigned the same piece of composite data as their value, each variable does *not* store a unique copy of the data (as it would if the data were primitive). Rather, multiple variables can point to one copy of the composite data. If the value of the data changes, all the variables that point to it reflect the updated value.

Let's see how this affects a practical application. When two variables refer to the same primitive data, each variable gets its own copy of the data. Here we assign the value 12 to the variable `x`:

```
var x = 12;
```

Now let's assign the value of `x` to a new variable, `y`:

```
var y = x;
```

As you can guess, `y` is now equal to 12. But `y` has its own copy of the value 12, distinct from the copy in `x`. If we change the value of `x`, the value of `y` is unaffected:

```
x = 15;
trace(x); // Displays: 15
trace(y); // Displays: 12
```

The value of `y` did not change when `x` changed because when we assigned `x` to `y`, `y` received its own copy of the number 12 (i.e., the *primitive* datum contained by `x`).

Now let's try the same thing with *composite* data. We'll create a new array with three elements and then assign that

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

array to the variable `x`:

```
var x = ["first element", 234, 18.5];
```

Now, just as we did before, we'll assign the value of `x` to `y`:

```
var y = x;
```

The value of `y` is now the same as the value of `x`. But what is the value of `x` ? Remember that because `x` refers to an array, which is a composite datum, the value of `x` is not literally the array `["first element", 234, 18.5]` but merely a reference to that datum. Hence, when we assign `x` to `y`, what's copied to `y` is not the array itself but the reference contained in `x` that points to the array. So, both `x` and `y` point to the same array, stored somewhere in memory.

If we change the array through the variable `x`, like this:

```
x[0] = "1st element";
```

the change is also reflected in `y`:

```
trace(y[0]);  // Displays: "1st element"
```

Similarly, if we modify the array through `y`, the change can be seen via `x`:

```
y[1] = "second element";
trace (x[1]);  // Displays: "second element"
```

To break the association, use the *slice( )* function to create an entirely new array:

```
var x = ["first element", 234, 18.5];
// Copy each element of x to a new array stored in y
var y = x.slice(0);
y[0] = "hi there";
trace(x[0]);  // Displays: "first element" (not "hi there")
trace(y[0]);  // Displays: "hi there" (not "first element")
```

Let's extend our example to see how primitive and composite data values are *compared*. Here we assign `x` and `y` an identical primitive value; then we compare the two variables:

```
x = 10;
y = 10;
trace(x =  = y);  // Displays: true
```

Because `x` and `y` contain primitive data, they are compared by value. In a value-based comparison, data is compared literally. The number 10 in `x` is considered equal to the number 10 in `y` because the numbers are made up of the same bytes.

Now, let's assign `x` and `y` identical versions of the same composite data and compare the two variables again:

```
x = [10, "hi", 5];
y = [10, "hi", 5];
trace(x =  = y);  // Displays: false
```

This time, `x` and `y` contain composite data, so they are compared by reference. The arrays we assigned to `x` and `y`

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

have the same byte values, but the variables `x` and `y` are not equal because they do not store a reference to the same composite datum. However, watch what happens when we copy the reference in `y` to `x`:

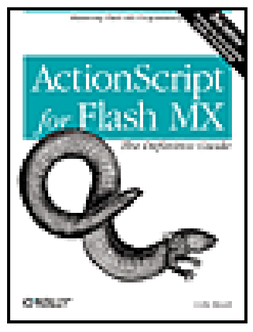
```
x = y;
trace(x =  = y);  // Displays: true
```

Now that the references are the same, the values are considered equal. Thus, the result of the comparison depends on the references in the variables, not the actual byte values of the arrays to which they point.

Primitive and composite data are also treated differently when passed to functions, as discussed under Section 9.8.3 in Chapter 9. Most notably, when a primitive variable is passed as an argument to a function, any changes to the datum within the function are not reflected in the original variable. However, when passing a composite variable, changes within the function do affect the original variable. That is, if you pass an integer variable `x` to a function, changes to the parameter within the function don't affect its original value in the calling routine. But if you pass an array `y` to a function, any changes to that array within the function will alter the original value of `y` outside the function (because changes to the array affect the data to which `y` points).

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir



ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

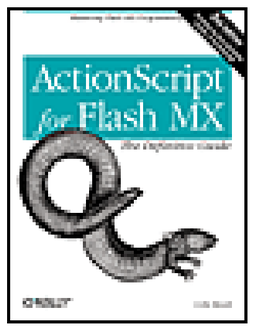
Chapter 3. Data and Datatypes

3.7 Onward!

We've introduced data in ActionScript, and we're ready for deeper study. In Chapter 4, we'll study the *number*, *string*, *boolean*, *undefined*, and *null* datatypes. In Chapter 5, we'll explore how to manipulate data using operators. In later chapters, we'll study the complex datatypes, such as *movieclips*, *arrays* and *objects*.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir



ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Part I: ActionScript Fundamentals

Chapter 4. Primitive Datatypes

Primitive data consists of simple characters or keywords, such as the numbers 0, 1, 2, 3, 4, 5, 6, 7, 8, and 9, or the strings "a", "b", and "c". As shown in Chapter 3, the primitive datatypes supported by ActionScript are *number*, *string*, *boolean*, *undefined*, and *null*. In this chapter, we'll explore how to define, examine, and change data of each type.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Chapter 4. Primitive Datatypes

4.1 The Number Type

Numbers are used for counting, mathematics, and keeping track of numeric properties in our movies (such as the current frame of a movie clip or its location on the Stage). Let's see how numbers are defined and manipulated in ActionScript.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Chapter 4. Primitive Datatypes

4.2 Integers and Floating-Point Numbers

Most programming languages distinguish between two kinds of numbers: *integers* and *floating-point numbers*. An integer is a whole number that has no fractional component. Integers can be positive, negative, or the number 0. Floating-point numbers (*floats* for short) can include a fractional value represented after a decimal point, as in 0.56, 199.99, and 3.14159. So, 1, 34523, -3, 0, and -9999999 are integers, but 223.45 and -0.56 are floats.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Chapter 4. Primitive Datatypes

4.3 Numeric Literals

We learned earlier that a *literal* is a direct representation of a single, fixed data value. The *number* type supports three kinds of literals: integer literals, floating-point literals, and special numeric values. The first two literal categories represent *real numbers* (numbers that have a fixed mathematical value); the third category comprises values that represent numeric concepts, such as infinity.

4.3.1 Integer Literals

Integer literals, such as 1, 2, 3, 99, and -200, must follow these rules:

- Integers may not contain a decimal point or fractional value.
- Integers must not exceed the minimum or maximum legal numeric values of ActionScript. See the `MIN_VALUE` and `MAX_VALUE` properties of the *Number* object in the *Language Reference* for a discussion of legal values.
- Base-10 integer numbers must not start with a leading zero (e.g., these are not valid base-10 integers: 002, 000023, and 05).

Not all integer values are base-10 (i.e., decimal) integers. ActionScript also supports base-8 (octal) and base-16 (hexadecimal) numeric literals. For a primer on decimal, octal, and hexadecimal numbers, see:

http://www.moock.org/asdg/technotes/basePrimer/

We use a leading zero to indicate an octal number. For example, to represent the octal number 723 in ActionScript, we use:

```
0723  // 467 in decimal (7*64 + 2*8 + 3*1)
```

To indicate a hexadecimal (*hex* for short) literal integer, we put 0x (or 0X) in front of the number, such as:

```
0x723  // 1827 in decimal (7*256 + 2*16 + 3*1)
0xFF   //  255 in decimal (15*16 + 15*1)
```

Hexadecimal numbers are often used to indicate color values, but most simple programs require only base-10 numbers. Be careful to remove unwanted leading zeros when converting strings to numbers, as shown in Example 4-1.

Example 4-1. Trim leading zeros

```
function trimZeros (theString) {
  while (theString.charAt(0) =  = "0" || theString.charAt(0) =  = " ") {
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

```
    theString = theString.substring(1, theString.length);
  }
  return theString;
}
testString = "00377";
trace(trimZeros(testString));  // Displays: "377"
```

4.3.2 Floating-Point Literals

Floating-point literals represent numbers containing fractional parts. A floating-point literal may contain some or all of these four components:

a base-10 integer
a decimal point (.)
a fractional portion (represented as a base-10 number)
an exponent

The first three components are pretty straightforward: in the number 3.14, "3" is the base-10 integer, "." is the decimal point, and "14" (i.e., 14 one-hundredths) is the fractional portion. But the fourth component (the exponent) requires a closer look.

To represent a very large positive or negative number as a float, we can append an exponent to a number using the letter E (or e). To determine the value of a number with an exponent, multiply the number by 10 to the power specified by the exponent. For example:

```
12e2    // 1200 (10 squared is 100, times 12 yields 1200)
143E-3  // 0.143 (10 to the power -3 is .001, times 143 yields 0.143)
```

You may recognize the format as standard *scientific notation* (a.k.a. *exponential notation*). If math isn't your strong point, here's an easy conversion tip: if the exponent is positive, move the decimal point that many places to the right; if the exponent is negative, move the decimal point that many places to the left.

ActionScript may return a number with an exponent as the result of a calculation, if the result is a very large or very small value. Note, however, that the exponent E (or e) is merely a notational convenience. If we want to raise a number to an arbitrary power, we use the built-in *Math.pow( )* function, which is documented in the ActionScript Language Reference.

4.3.2.1 Floating-point precision

Flash uses *double-precision* floating-point numbers, which offer precision to about 15 significant digits. (Any leading zeros, trailing zeros, and/or exponents are not counted as part of the 15 digits.) This means that Flash can represent the number 123456789012345, but not 1234567890123456. The precision doesn't limit how big a number can get, only how precisely a number can be represented; 2e16 is a bigger number than 123456789012345, but it employs only one significant digit.

Occasionally, ActionScript calculations are rounded in undesirable ways, producing numbers such as 0.14300000000000001 instead of 0.143. This happens because computers convert numbers of any base to an internal binary representation, which can lead to nonterminating fractions in binary (much like 0.3333333 in decimal). Computers have only finite precision, so they cannot perfectly represent nonterminating fractions. In order to compensate for the minute discrepancy, you should round your numbers manually if the difference will adversely affect the behavior of your code. For example, here we round `myNumber` to three decimal places:

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

```
myNumber = Math.round(myNumber * 1000) / 1000;
```

And here's a reusable function to round any number to an arbitrary number of decimal places:

```
function roundNum (theNumber, decPlaces) {
  if (decPlaces >= 0) {
    var temp = Math.pow(10, decPlaces);
    return Math.round(theNumber * temp) / temp;
  }
}
// Round a number to two decimal places
trace(roundNum(1.12645, 2));  // Displays: 1.13
```

4.3.3 Special Values of the Number Datatype

Integer and floating-point literals account for nearly all the legal values of the *number* datatype, but there are special keyword values that represent the following numeric concepts: Not-a-Number, Minimum Allowed Value, Maximum Allowed Value, Infinity, and Negative Infinity.

Each of the special values can be assigned to variables and properties or used in literal expressions, just like any other numeric literal. More often than not, though, the special numeric values are returned by the interpreter as the result of some expression evaluation.

4.3.3.1 Not-a-Number: NaN

Occasionally, a mathematical computation or an attempted datatype conversion results in a value that simply is not a number. For example, 0/0 is an impossible calculation, and the following expression can't be converted to a finite number:

```
23 - "go ahead and try!"
```

In order to accommodate data that is of the *number* type but is not a calculable number, ActionScript provides the `NaN` keyword value. Though `NaN` doesn't represent a calculable number, it is still a legal value of the *number* type, as demonstrated by the following code:

```
x = 0/0;
trace(x);          // Displays: NaN
trace(typeof x);   // Displays: "number"
```

Since `NaN` is not a finite numeric value, it never compares as equal to itself. If two variables hold the value `NaN`, they are considered not equal (though they may seem equal to us). To work around this problem, we use the built-in function *isNaN( )* to check whether a variable contains the `NaN` value:

```
x = 12 - "this doesn't make much sense";  // x is now NaN
trace(isNaN(x));                          // Displays: true
```

4.3.3.2 Minimum and maximum allowed values: MIN_VALUE and MAX_VALUE

ActionScript can represent a broad, but not unlimited, range of numbers. The maximum allowable value is 1.7976931348623e+308, and the minimum allowed value is 4.94065645841247e-324. Obviously, these numbers are cumbersome, so ActionScript offers the special values `Number.MAX_VALUE` and `Number.MIN_VALUE` for convenience.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

`Number.MAX_VALUE` comes in handy when we are checking to see if a calculation results in a representable number:

```
z = x*y;
if (z <= Number.MAX_VALUE && z >= -Number.MAX_VALUE) {
  // Number is legal
}
```

Note that `Number.MIN_VALUE` is the smallest *positive* value allowed, not the most *negative* value. The "largest" negative legal value is `-Number.MAX_VALUE`.

4.3.3.3 Infinity and negative infinity: Infinity and -Infinity

If a calculation results in a value larger than `Number.MAX_VALUE`, ActionScript uses the keyword `Infinity` to represent the result of the calculation (this is known as an *overflow* condition). Similarly, if a calculation results in a value more negative than the "largest" allowable negative value, ActionScript uses `-Infinity` to represent the result (this is known as an *underflow* condition). `Infinity` and `-Infinity` can also be used directly as literal numeric expressions.

4.3.3.4 Irrational numbers

In addition to the special numeric values—`NaN`, `Infinity`, `-Infinity`, `Number.MAX_VALUE`, and `Number.MIN_VALUE`—ActionScript provides convenient access to mathematical constants via the *Math* object. For example:

```
Math.E        // The value of e, the base of the natural logarithm
Math.LN10     // Natural logarithm of 10
Math.LN2      // Natural logarithm of 2
Math.LOG10E   // Base-10 logarithm of e
Math.LOG2E    // Base-2 logarithm of e
Math.PI       // Pi (i.e., 3.1415926...)
Math.SQRT1_2  // Square root of 1/2
Math.SQRT2    // Square root of 2 (i.e., 1.4142135...)
```

These constants are simply shorthand forms of floating-point values that approximate commonly used irrational numbers. (Irrational numbers, by definition, are numbers that cannot be represented by a simple fraction, such as 1/3. They should not be confused with `NaN`, `Infinity`, etc.) You can use these irrational numbers just as you would any other object property:

```
area = Math.PI*(radius*radius);
```

For a complete list of supported constants, see the *Math* object in the *Language Reference*.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Chapter 4. Primitive Datatypes

4.4 Working with Numbers

We can manipulate numbers by combining them with operators to form mathematical expressions and by calling built-in functions to perform complex mathematical operations.

4.4.1 Using Operators

Basic arithmetic—addition, subtraction, multiplication, and division—is accomplished using the +, -, *, and / operators. Operators can be used with any numeric literals or data containers such as variables. Mathematical expressions are evaluated in the order determined by the *precedence* of the operators, as shown in Table 5-1. For example, multiplication is performed before addition. All of these are legitimate uses of mathematical operators:

```
x = 3 * 5;              // Assign the value 15 to x
x = 1 + 2 - 3 / 4;      // Assign the value 2.25 to x
x = 56;
y = 4 * 6 + x;          // Assign the value 80 to y
y = x + (x * x) / x;    // Assign the value 112 to y
```

4.4.2 Built-in Mathematical Functions

To perform advanced math, we use the built-in mathematical functions of the *Math* object. For example:

```
Math.abs(x)         // Absolute value of x
Math.min(x, y)      // The smaller of the values x and y
Math.pow(x, y)      // x raised to the power y
Math.round(x)       // x rounded to the nearest integer
```

These math functions return values that we use in expressions, just as we use literal numbers. For example, suppose we want to simulate a six-sided die. We can use the *Math.random( )* function to retrieve a random float between 0 and 0.999 . . . :

```
dieRoll = Math.random();
```

Then, we multiply that value by 6, giving us a float between 0 and 5.999, to which we add 1:

```
dieRoll = dieRoll * 6 + 1;  // Sets dieRoll to a number between 1 and 6.999
```

Finally, we use the *floor( )* function to round our number down to the closest integer:

```
dieRoll = Math.floor(dieRoll);  // Sets dieRoll to a number between 1 and 6
```

Condensed into a single expression, our die roll calculation looks like this:

```
// Sets dieRoll to a number between 1 and 6 inclusive
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

```
dieRoll = Math.floor(Math.random() * 6 + 1);
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Chapter 4. Primitive Datatypes

4.5 The String Type

String is the datatype used for textual data (letters, punctuation marks, and other characters). A string literal is any combinatic characters enclosed in single or double quotation marks:

```
"asdfksldfsdfeoif"  // A frustrated string
"greetings"         // A friendly string
"unity@moock.org"   // A self-promotional string
"123"               // It may look like a number, but it's a string
'singles'           // Single quotes are acceptable too
```

Before we see how to form string literals, let's examine which characters are permitted in strings.

4.5.1 Character Encoding

Like all computer data, text characters are stored internally using a numeric code. They are *encoded* for storage and *decoded* f using a *character set* , which *maps* (i.e., relates) characters to their numeric codes. Character sets vary for different languages alphabets. Originally, most Western applications used some derivative of ASCII, a standard character set that includes only 1 characters—the English alphabet, numbers, basic punctuation marks, and special *control characters* . In time, Western applica supported a family of character sets known collectively as ISO-8859, an extension of ASCII. Each of the ISO-8859 character encodes the standard Latin alphabet (*A* to *Z*), plus a varying set of letters needed in the target languages. Flash Player 5, for e uses ISO-8859-1, also known as *Latin 1* , as its primary character map (it also supports a second character set, called *Shift-JIS* Japanese characters).

The Latin 1 character set accommodates most Western European languages—French, German, Italian, Spanish, Portuguese, a on—but not languages such as Greek, Turkish, Slavic, and Russian. In order to support the full range of the world's languages Player 6 uses *Unicode* , the preferred international standard for character encoding, which can map a million or more characte Flash Player 6 can display most characters supported by Unicode, regardless of the regional settings, provided that the approp font is available on the end user's system. However, Flash Player 6 does not support right-to-left or bidirectional scripts, such and Hebrew.

Appendix B lists each character's Unicode *code point* , which is the character's numeric position in the Unicode set. Later, we how to use code points to manipulate characters in our scripts and to display Unicode characters. For more information on Un the resources cited in Appendix A.

4.5.2 String Literals

The most common way to create a string is to put either single or double quotation marks around a group of characters:

```
"hello"
'Nice night for a walk.'
"The equation is 12 + 4 = 16, which programmers see as 12 + 4 =  = 16."
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

If we use a double quotation mark to start a string, we must end it with a double quotation mark as well. Likewise, if we use a quotation mark to start a string, we must end that string with a single quotation mark. However, a double-quoted string may c single-quoted characters and vice versa. These strings, for example, contain legal uses of single and double quotes:

```
"Nice night, isn't it?"                 // Single apostrophe inside double quote
'I said, "What a pleasant evening!"'  // Double quotes inside single quotes
```

4.5.2.1 The empty string

The shortest possible string is the *empty string* , a string with no characters:

```
""
''
```

The empty string is handy when we're trying to detect whether a variable contains a usable string value or not. Here we check the input text field, `firstName_txt` , has a usable value:

```
if (firstName_txt.text =  = "") {
  trace("You forgot to enter your name!");
}
```

The empty string is not considered equal to one or more space characters (e.g., `""` is not the same as `" "`). To check if a strii least one usable character, we revise our code to detect any character above Unicode code point 32 (everything below code pc either a control character or a whitespace character), as shown in Example 4-2.

Example 4-2. A custom String.isEmpty( ) method

```
// Checks whether the string contains any characters above Unicode 32.
String.prototype.isEmpty = function ( ) {
  // If a useful character is found...
  for (var i=0; i < this.length; i++) {
    if (this.charCodeAt(i) > 32) {
       // The string is not "empty."
       return false;
    }
  }

  // No useful characters were found. The string is "empty."
  return true;
}

// Now use the new method to test if our firstName_txt text field has a useful
if (firstName_txt.text.isEmpty( )) {
  trace("You forgot to enter your name!");
} else {
  trace("Welcome, " + firstName_txt.text);
}
```

4.5.2.2 Escape sequences

We have seen that single quotes (') may be used inside double-quoted literals, and double quotes (") may be used inside sing

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

literals. But what if we want to use both? For example:

```
'I remarked "Nice night, isn't it?"'
```

As is, this line of code causes an error because the ActionScript compiler thinks that the string literal ends with the apostrophe word "isn't." The compiler reads it as:

```
'I remarked "Nice night, isn'  // The rest is considered unintelligible garbag
```

To use the single quote inside a string literal delimited by single quotes, we must use an *escape sequence* . An escape sequenc represents a literal string value using a backslash (\), followed by the desired character or a code that represents the character. escape sequences for single and double quotes are:

```
\'
\"
```

So, our cordial evening greeting, properly expressed as a string literal, should be:

```
'I remarked "Nice night, isn\'t it?"'  // Escape the apostrophe!
```

Other escape sequences, which can be used to represent various special or reserved characters, are listed in Table 4-1.

Table 4-1. ActionScript escape sequences

Escape sequence	Meaning
`\b`	Backspace character (ASCII 8)
`\f`	Form feed character (ASCII 12)
`\n`	Newline character; causes a line break (ASCII 10)
`\r`	Carriage return (CR) character; causes a line break (ASCII 13)
`\t`	Tab character (ASCII 9)
`\'`	Single quotation mark
`\"`	Double quotation mark
`\\`	Backslash character; necessary when using backslash as a literal character, to prevent `\` from being interpret beginning of an escape sequence

4.5.2.3 Unicode escape sequences

Not all Unicode characters are accessible from a keyboard. In order to include inaccessible characters in a string, we use Unic escape sequences.

A Unicode escape sequence starts with a backslash and a *lowercase u* (i.e., `\u`) followed by a four-digit hex number that cor to the Unicode character's code point, such as:

```
\u0040  // The @ sign
\u00A9  // The copyright symbol
\u0041  // The capital letter "A"
\u2014  // The em dash
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

A *code point* is a unique identification number that is assigned to each character in the Unicode character set. See Appendix I list of the Unicode code points for Latin 1. Code points for other languages can be found in the character charts at the Unicod Consortium site:

http://www.unicode.org/charts

If you have trouble finding a character amongst the thousands of code points, consult the Unicode Consortium's helpful sugge

http://www.unicode.org/unicode/standard/where/

To escape characters from the Latin 1 character set only, we can use a short form for the standard Unicode escape sequence. T form consists of the prefix `\x` followed by a two-digit hexadecimal number that represents the Latin 1 code point of the chara Since Latin 1 code points are the same as the first 256 Unicode code points, you can still use the reference chart in Appendix simply remove the `u00` , as in the following examples:

```
\u0040  // Unicode escape sequence
\x40    // \x shortcut form
\u00A9  // Unicode...
\xA9    // ...you get the idea
```

In addition to using Unicode escape sequences, we can insert any character into a string via the built-in *fromCharCode( )* func described later in this chapter under Section 4.6.9. The *fromCharCode( )* function accepts a code point as any number-yieldi expression, whereas an escape sequence requires a numeric literal.

> Flash Player 5 supported Unicode-style escape sequences, but it did not support Unicode. A Unicode escaj sequence in Flash 5 could be used to specify code points in the Latin 1 or Shift-JIS character sets only, not Unicode code points. Inserting Unicode code points will not yield the correct character in Flash Player 5 o in *.swf* files exported in Flash 5 format.

4.5.3 Entering Multilingual and Special Characters

Though Flash Player 6 fully supports Unicode, the Flash MX authoring tool does not. The Flash MX authoring tool allows en characters from the Latin 1, Shift-JIS, and MacRoman character sets. Other character sets, combined with international keybo OS combinations may pose display problems. Hence, the most flexible way to enter Unicode text in Flash MX is to load it frc external Unicode-formatted text file. On Windows, Notepad, Word, and WordPad all support saving as Unicode. On Macinto TextEdit supports saving as Unicode. The simplest way to import a Unicode text file is to use the `#include` directive in co the special `//!-- UTF8` comment, which effectively copies and pastes the contents of the file into a Flash movie at compil information on using `#include` with Unicode, see the `#include` directive in the *Language Reference* . External Unicod also be stored as XML or URL-encoded variables. For information on loading external XML or variable files, see the *XML.lo LoadVars.load( )* methods in the ActionScript Language Reference.

Note that the specific encoding format supported by `#include` and *LoadVars* is UTF-8, while the encoding formats suppo *XML* are UTF-8, UTF16-BE, and UTF-16LE (i.e., either big-endian or little-endian). When a file is UTF-16 encoded, it is ex to start with a byte order marker (BOM) indicating whether the encoding is big-endian or little-endian. Most text editors add t automatically. When no BOM is present in a file, the encoding is assumed to be UTF-8. When in doubt, you should use UTF- encoding, where byte order is not an issue. For more information on UTFs and BOMs, see:

http://www.unicode.org/unicode/faq/utf_bom.html

To include small amounts of Unicode directly within the Flash MX authoring tool, use either the standard Unicode hex escap sequence (described earlier in this chapter) or *String.fromCharCode( )* (described later in this chapter).

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

For example, the following code creates a global variable, `euro` , that contains the euro sign character, and then inserts that c into the `price_txt` text field for display.

```
_global.euro = "\u20AC";
this.createTextField("price_txt", 1, 100, 100, 200, 20);
price_txt.text = "99 " + euro;
```

In Flash Player 5, all Unicode (\u) escape sequences must reference a code point in the Latin 1 or Shift-JIS character sets; oth Unicode characters will not display correctly. For example, in Flash Player 5, the code in our example would not display the € character correctly.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Chapter 4. Primitive Datatypes

4.6 Working with Strings

By manipulating strings we can program anything, from a user-input validator to a word-scramble game. With a little ingenui make neat visual text effects, text tickers, and other fun stuff.

We can manipulate strings with both operators and built-in functions. String operators can join multiple strings together or co characters of two strings. Built-in functions can examine a string's properties and contents, extract a portion of a string, check character's code point, create a character from a code point, change the case of the characters in a string, and even turn a strin variable or property name.

4.6.1 Joining Strings Together

Joining strings together (creating a new string from two or more strings) is called *concatenation* . As we've seen, we can conc two strings with the plus operator (`+`), like this:

```
"Macromedia" + "Flash"
```

This line of code yields the single string value "MacromediaFlash". Oops! We forgot to put a space between the words. To ad space, we can insert it within the quotes that define one of the strings, such as:

```
"Macromedia " + "Flash"  // Yields "Macromedia Flash"
```

But that's not always practical. In most cases, we don't want to add a space to a company or a product name. So instead, we jo strings together, the middle one of which is simply an empty space:

```
"Macromedia" + " " + "Flash"   // Also yields "Macromedia Flash"
```

Note that the space character is not the same as the empty string we discussed earlier, because the empty string has no charact between the quotes.

Ordinarily, you wouldn't concatenate literal strings, because you could more easily write the result as one string—"Macromed Flash"—in the first place. As a more realistic example, we can concatenate variables that contain string data. Consider the fol code:

```
var company = "Macromedia";
var product = "Flash";
// Set the variable sectionTitle to "Macromedia Flash"
var sectionTitle = company + " " + product;
```

In lines 1 and 2, we store string values in variables. Then, we join these values together with a space. Two of our string value contained in variables, and one (the space) is a string literal. This is not a problem; it happens all the time.

Occasionally, we'll want to append characters onto an existing string. For example, we can create a custom welcome message site like this:

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

```
// Hard-code the userName for this example.
// (Normally it would be retrieved via an input text field or from a database.
var userName = "Karsten";
var greeting = "Welcome to GreatSite.com";  // Our welcome message
greeting = greeting + " " + userName;       // Add user's name to welcome mess
```

This gets the job done, but notice that we have to refer to `greeting` twice in line 2. To be more efficient, we can use the `+=` operator, which appends the string on its right to the string variable on the left:

```
var greeting = "Welcome to GreatSite.com";  // Our welcome message
greeting += (" " + username);               // Add user's name to welcome mess
```

You may encounter the deprecated Flash 4 string concatenation operators `&` and *add* when examining legacy code. The `+` ope should be used in Flash 5 and later. See Appendix C for more information on backward compatibility.

4.6.1.1 The concat( ) function

The *concat( )* function appends characters to a string, similar to `+=` . Because *concat( )* is a function, it uses the dot operator, l

```
var product = "Macromedia".concat(" Flash");
var sentence = "How are you";
var question = sentence.concat("?")
```

Take heed, though—unlike `+=` , the *concat( )* function does not alter the string that it is applied to; it merely returns the conca string value. In order to make use of that value, we must assign it to a variable or other data container. Study the following co so you'll understand the difference between `+=` and *concat( )* :

```
// Here we mistakenly forget to store the return of concat( ) in a variable.
var userName = "Karsten";
var greeting = "Welcome to GreatSite.com";
greeting.concat(userName);
trace(greeting);  // Displays "Welcome to GreatSite.com"
                  // The value of greeting was unaffected by concat( )
// Here we properly store the return of concat( ) in a variable.
finalGreeting = greeting.concat(" " + userName);
```

The *concat( )* function also accepts multiple arguments (that is, it can combine multiple comma-separated strings into one stri

```
var userName = "Karsten";
// Sets finalGreeting to "Welcome to GreatSite.com, Karsten!"
finalGreeting = greeting.concat(" ", username, "!");
```

This code is the same as:

```
finalGreeting = greeting;
finalGreeting += " " + userName + "!";
```

4.6.2 Comparing Strings

To check whether two strings are the same, we use the equality (`= =`) and inequality (`!=`) operators. We often compare stri executing code based on a condition. For example, if a user enters a password, we need to compare his input string with the a password. The result of our comparison governs the behavior of our code, such as whether to grant access to the password-res

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

functionality.

4.6.2.1 Using the equality (= =) and inequality (!=) operators

The equality operator takes two operands—one on its left and one on its right. For string comparisons, the operands can be str or any variable, array element, object property, or expression that can be converted to a string:

```
"hello" =  = "goodbye"        // Compare two string literals
userGuess =  = "fat-cheeks"   // Compare a variable with a string
userGuess =  = password       // Compare two variables
```

Although Flash 4 used different operators for its string and nonstring comparisons, in Flash 5 and later, the `= =` and `!=` ope also be used to compare numbers and other datatypes (see Appendix C). We limit the discussion here to strings. If the operand right has the exact same characters in the exact same order as the operand on the left, the two strings are considered equal, and is the Boolean value `true` .

Two individual characters in ActionScript are considered equal only if they have the same Unicode code point. This means th and lowercase letters (which have different code points) are not considered equal. Likewise, letters with and without diacritica such as *u* and *u* , have different code points and are not considered equal. Furthermore, ActionScript does not consider any mu character sequences to be equal to the corresponding single-glyph character. For example, *n*~ is not considered equal to *n* (thc two might be considered equal in other languages).

The following comparisons all evaluate to `false` :

```
"olive-orange" =  = "olive orange"  // Not equal
"nighttime" =  = "night time"       // Not equal
"Day 1" =  = "day 1"                // Not equal
"n~" =  = "n"                       // Not equal
"u" =  = "u"                        // Not equal
```

Because string comparisons result in the Boolean value `true` or `false` , we can use them as test expressions within conditi statements and loops, like this:

```
if (userGuess =  = password) {
  this.gotoAndStop("classifiedContent");
} else {
  this.gotoAndStop("entryFailed");
}
```

If the expression *(userGuess = = password)* is `true` , then the *this.gotoAndStop("classifiedContent");* statement will be exec expression is `false` , the classified content is skipped, and the user is sent to a frame where an error message is displayed.

We'll learn more about Boolean values later in this chapter, and we'll learn about conditional statements in Chapter 7.

To check whether two strings are not equal, we use the *inequality* operator, which yields a result opposite to the equality oper example, the following expressions represent the values `false` and `true` , respectively:

```
"Jane" != "Jane"  // false, because the two strings are equal
"Jane" != "Biz"   // true, because the strings are different
```

Here we use the inequality operator to take some action only if two strings are not equal:

```
if (userGender != "boy") {
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

```
  // Girls-only code goes here...
}
```

You may encounter the deprecated `eq` and `ne` operators, used for string equality and inequality comparisons in Flash 4, whe examining legacy code. The `==` and `!=` operators should be used in Flash 5 and later. See Chapter 5 for coverage of the *st equality* and *strict inequality* operators, which test equality without performing automatic datatype conversion.

4.6.2.2 Character order and alphabetic comparisons

We can also compare two strings on a character-order basis. We saw earlier that each character has a numeric code point assi and that these code points are ordered numerically in a character set (in ActionScript's case, that character set is Unicode). W check which character comes first in the order using the *comparison* operators: greater than (`>`), greater than or equal to (`>=` (`<`), and less than or equal to (`<=`). Each comparison operator compares two operands:

```
"a" < "b"       // true
"2" > "&"       // true
"r" <= "R"      // false
"$" >= "@"      // false
```

Much like equality and inequality expressions, comparison expressions yield a Boolean value, `true` or `false`, depending relationship of the operands. Each operand can be anything that yields a string value. The comparison operators can also be u compare other datatypes, such as numbers, but we limit our discussion here to string operands.

Since the characters *A* to *Z* and *a* to *z* are grouped in alphabetic sequence in both Unicode and Latin 1, we frequently use cha order comparisons to determine which of two letters comes first alphabetically in the Latin alphabet. Note, however, that all u letters come before all lowercase letters in the Latin 1 character set. If we forget this, we're in for some surprising results:

```
"Z" < "a"       // Evaluates to true
"z" < "a"       // Evaluates to false
"Cow" < "bird"  // Evaluates to true
```

ActionScript does not use the Unicode Collation Algorithm (UCA) when sorting characters; characters are compared on a stri point basis. The UCA outlines the rules for comparing international characters using regional alphabetization and accounting diacritical marks, and multicharacter sequences. For more information, see:

http://www.unicode.org/unicode/reports/tr10

Here's a closer look at each comparison operator; in the following descriptions, the *comparison character* is defined as the firs nonidentical character found in the two operands:

Greater than (`>`)

Yields `true` if the comparison character of the left operand appears later in the Unicode character order than the com character of the right operand. If the two operands are identical, `>` returns `false`:

```
"b" > "a"       // true
"a" > "b"       // false
"ab" > "ac"     // false (the second character is the comparison character)
"abc" > "abc"   // false (the strings are identical)
"ab" > "a"      // true (b is the comparison character)
"A" > "a"       // false ("A" comes before "a" in the character order)
```

Greater than or equal to (`>=`)

Yields `true` if the comparison character of the left operand appears later in the character order than the comparison c

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

the right operand or if the two operands are identical:

```
"b" >= "a"  // true
"b" >= "b"  // true
"b" >= "c"  // false
"A" >= "a"  // false ("A" and "a" occupy different code points)
```

Less than (<)

Yields `true` if the comparison character of the left operand appears earlier in the Unicode character order than the co character of the right operand. If the two operands are identical, `<` returns `false` :

```
"a" < "b"       // true
"b" < "a"       // false
"az" < "aa"     // false (the second character is the comparison character)
```

Less than or equal to (<=)

Yields `true` if the comparison character of the left operand appears earlier in the character order than the comparison of the right operand or if the two operands are identical:

```
"a" <= "b"  // true
"a" <= "a"  // true
"z" <= "a"  // false
```

To determine which of two nonalphabetic characters comes first in the Latin 1 character order (which follows the Unicode or consult Appendix B. To determine the Unicode code point for any character, see http://www.unicode.org/charts.

The following example checks whether a character is a letter from the Latin alphabet (as opposed to a number, punctuation m other symbol):

```
var theChar = "w";
if ((theChar >= "A" && theChar <= "Z") || (theChar >= "a" && theChar <= "z"))
  trace("The character is in the Latin alphabet.");
}
```

Notice how the logical *OR* operator (`||`) lets us check two conditions at once. We'll study *OR* in Chapter 5.

You may encounter the deprecated `gt` , `ge` , `lt` , and `le` operators, used for string comparison in Flash 4, when examining code. The `>` , `>=` , `<` , and `<=` operators should be used in Flash 5 and later.

4.6.3 Using Built-in String Functions

With the exception of the *concat( )* function, every tool we've used with strings so far has been an operator. Now we'll see hov built-in functions and properties to perform more advanced string manipulation.

To execute a built-in function on a string, we must perform a *function call* , which takes the form:

```
string.functionName(arguments)
```

For example, here we execute the *charAt( )* function on `myString` :

```
myString.charAt(2)
```

Many string functions return a new string derived from the original string. We assign these return values to variables or objec properties for future use, such as:

```
thirdCharacter = myString.charAt(2);
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

4.6.4 Character Indexing

Many of the string functions make use of a character's *index* —its numeric position relative to the string's first character, start not 1. The first character is numbered 0, the second is numbered 1, the third is numbered 2, and so on. For example, in the stri "red", the *r* is at index 0, the *e* is at index 1, and the *d* is at index 2.

Using character indexes we identify portions of a string. We may, for example, instruct the interpreter to "Get the characters f index 3 to index 7," or we may ask it, "What character is at index 5?"

4.6.5 Examining Strings

We can inspect and search within strings using the built-in `length` property or the *charAt( )* , *indexOf( )* , and *lastIndexOf(* functions.

4.6.5.1 The length property

The `length` property tells us how many characters are in a string. Because it is a property, not a function, we don't use pare or arguments when referring to it (the older Flash 4 *length( )* function was deprecated in Flash 5). Here we see the `length` strings:

```
"Flash".length          // length is 5
"skip intro".length     // length is 10 (spaces count as characters)
"".length               // The empty string contains 0 characters
var axiom = "all that glisters will one day be obsolete";
axiom.length            // 42
```

Because character indexes start at 0 (i.e., they are *zero-relative*), the index of the last character is always equal to the `lengt` string minus one.

A string's `length` property can be read but not set. We can't make a string longer by doing this:

```
axiom.length = 100;  // Nice try, but it ain't gonna work
```

4.6.5.2 The charAt( ) function

We can determine the character at any index within a string using the *charAt( )* function, which takes the form:

```
string.charAt(index)
```

where *`string`* may be any literal string value or an identifier that contains a string; *`index`* is an integer, or an expressior resolves to an integer, that indicates the position of the character we want to retrieve. The value of *`index`* should be betwee *`string`* `.length - 1` . If *`index`* does not fall in that range, the empty string is returned. Here are some examples:

```
"It is 10:34 pm".charAt(1)  // Returns "t", the second character
var country = "Canada";
country.charAt(2);          // Returns "n", the third character
var x = 4;
fifthLetter = country.charAt(x);                    // fifthLetter is "d"
lastLetter = country.charAt(country.length - 1);  // lastLetter is "a"
```

4.6.5.3 The indexOf( ) function

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

We use the *indexOf( )* function to search for characters in a string. If the string we're searching contains our search sequence, returns the index (i.e., position) of the sequence's first occurrence in the string. Otherwise, it returns the value -1. The general *indexOf( )* is:

```
string.indexOf(character_sequence, start_index)
```

where `string` is any literal string value or an identifier that contains a string; `character_sequence` is the string fo we're searching, which may be a string literal or an identifier that contains a string; and `start_index` is the zero-relative position of the search. If `start_index` is omitted, the search starts at the beginning of `string` .

Let's use *indexOf( )* to check whether a string contains the character *W* :

```
"GWEN!".indexOf("W");  // Returns 1
```

Yup, *W* is the second character in "GWEN!", so we get 1, the index of the *W* character. Remember, character indexes start a second character's index is 1.

What happens if we search for the lowercase character *w* ? Let's see:

```
"GWEN!".indexOf("w");  // Returns -1
```

There is no *w* in "GWEN!", so *indexOf( )* returns -1. The upper- and lowercase versions of a letter are different characters, ar *indexOf( )* is case sensitive!

Now let's make sure that there's an @ sign in an email address:

```
var email = "daniella2dancethenightaway.ca";  // Oops, someone forgot to press
// If there's no @ sign, warn the user via the formStatus_txt text field
if (email.indexOf("@") =  = -1) {
  formStatus_txt.text = "The email address is not valid.";
}
```

We don't have to limit our searches to single characters. We can also search for an entire character sequence in a string. Let's "Canada" in the address of a company:

```
var iceAddress = "St. Clair Avenue, Toronto, Ontario, Canada";
iceAddress.indexOf("Canada");  // Returns 36, the index of the "C" in "Canada"
```

Notice that *indexOf( )* returns the position of the first character in "Canada". Now let's compare the return value of *iceAddress.indexOf("Canada")* to -1, and assign the result to a variable that stores the nationality of the company:

```
var isCanadian = iceAddress.indexOf("Canada") != -1;
```

The value of *iceAddress.indexOf("Canada") != -1* will be `true` if *iceAddress.indexOf("Canada")* does not equal -1 (if "Can found) and `false` if *iceAddress.indexOf("Canada")* does equal -1 (if "Canada" is not found). We then assign that Boolean v the variable `isCanadian` , which we can use to create a country-specific mailing form for North America:

```
if (isCanadian) {
  mailDesc = "Please enter your postal code.";
} else {
  mailDesc = "Please enter your ZIP code.";
}
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

The *indexOf( )* function can also help us determine which part of a string we need to extract. We'll see how that works when w about the *substring( )* function.

4.6.5.4 The lastIndexOf( ) Function

We just saw that the *indexOf( )* function returns the location of a character sequence's *first* occurrence in a string. The *lastInde* function returns the location of a character sequence's *last* occurrence in a string, or -1 if the sequence isn't found. The general *lastIndexOf( )* is just like that of *indexOf( )* :

```
string.lastIndexOf(character_sequence, start_index)
```

The only difference is that since *lastIndexOf( )* searches a string backward, `start_index` refers to the rightmost characte included in our search (not the leftmost). If `start_index` is omitted, it defaults to `string .length - 1` (the last ch the string).

For example:

```
paradox = "pain is pleasure, pleasure is pain";
paradox.lastIndexOf("pain");     // Returns 30; indexOf( ) would return 0
```

The following returns 0 (the index of the first occurrence of the word "pain"), because we started the backward search before second occurrence of "pain":

```
paradox.lastIndexOf("pain",29);  // Returns 0
```

4.6.5.5 No regular expressions

Note that regular expressions (a powerful tool used to recognize patterns in textual data) are not supported in ActionScript by However, developer Pavils Jurjans has made a custom RegExp object available at http://www.jurjans.lv/flash/RegExp.html.

4.6.6 Retrieving Portions of Strings

Sometimes a long string contains a sequence of characters that we'd like to access more conveniently. In the string "Steven Si Mumby", for example, we may want to extract the last name, "Mumby". To extract a shorter string (or *substring*), we use one functions: *substring( )* , *substr( )* , *splice( )* , or *split( )* . We'll cover how each of them work before discussing performance considerations.

4.6.6.1 The substring( ) function

We use *substring( )* to retrieve a sequence of characters from a string, based on starting and ending character indexes. The *sul* function takes the following form:

```
string.substring(start_index, end_index)
```

where `string` is any literal string value or an identifier that contains a string, `start_index` is the index of the first ch include in the substring, and `end_index` is the character *after* the last character we want in our substring. If not provided, `end_index` defaults to `string .length` . Hence:

```
var fullName = "Steven Sid Mumby";
middleName = fullName.substring(7, 10);  // Assigns "Sid" to middleName
firstName = fullName.substring(0, 6);    // Assigns "Steven" to firstName
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

```
lastName = fullName.substring(11);       // Assigns "Mumby" to lastName
```

In reality, we wouldn't know where the first name, middle name, and last name begin and end, so we'd typically look for some , such as the space character to help us guess where the word breaks are (see the discussion of the *split( )* function for an easie do this). Here we search for the last space in `fullName` and assume that the remainder of the string following it is the user's name:

```
fullName = "Steven Sid Mumby";
lastSpace = fullName.lastIndexOf(" ");  // Returns 10
// Characters from lastSpace+1 to the string's end are presumably the last nam
lastName = fullName.substring(lastSpace+1);
trace ("Hello Mr. " + lastName);
```

If `start_index` is greater than `end_index` , the two arguments are swapped automatically before the function execu Although the following function invocations yield the same result, you shouldn't make a habit of using *substring( )* with the ir reversed because it makes your code harder to understand:

```
fullName.substring(4, 6);  // Returns "en"
fullName.substring(6, 4);  // Returns "en"
```

4.6.6.2 The substr( ) function

The *substr( )* function extracts a sequence of characters from a string using a starting index and a length (in contrast to *substri* hich uses starting and ending indexes). The general form of *substr( )* is:

```
string.substr(start_index, subLength)
```

where `string` is, as usual, any literal string value or an identifier that contains a string; `start_index` is the first chara include in the substring; and `subLength` specifies how many characters should be included in our string, starting at `start_index` and counting to the right. If `subLength` is omitted, the substring starts at `start_index` and ends v last character in the original string. Here are some examples:

```
var fullName = "Steven Sid Mumby";
middleName = fullName.substr(7, 3);    // Assigns "Sid" to middleName
firstName = fullName.substr(0, 6);     // Assigns "Steven" to firstName
lastName = fullName.substr(11);        // Assigns "Mumby" to lastName
```

The `start_index` can be specified relative to the *end* of a string by using a negative number. The last character is -1, the to-last character is -2, and so on. So, the preceding three *substr( )* examples could be written as:

```
middleName = fullName.substr(-9, 3);  // Assigns "Sid" to middleName
firstName = fullName.substr(-16, 6);  // Assigns "Steven" to firstName
lastName = fullName.substr(-5);       // Assigns "Mumby" to lastName
```

A negative *subLength* , however, is not allowed.

4.6.6.3 The slice( ) function

Like *substring( )* , the *slice( )* function retrieves a sequence of characters from a string, based on starting and ending character While *substring( )* can specify only the indexes relative to the *beginning* of the original string, *slice( )* can specify them relativ string's beginning or end.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

The *slice( )* function takes the following form:

```
string.slice(start_index, end_index)
```

where `string` is any literal string value or an identifier that contains a string and `start_index` is the first character to the substring. If `start_index` is a positive integer, it is a normal character index; if `start_index` is a negative integ equivalent character index is determined by counting backward from the end of the string (i.e., `string .length + start_index`). Finally, `end_index` is the character *after* the last character we want in our substring. If `end_inde` provided, it defaults to `string .length` . If `end_index` is negative, the equivalent character index is determined by backward from the end of the string (i.e., `string .length + end_index`).

Using nonnegative indexes with *slice( )* works just like *substring( )* . When using negative indexes, remember that you are not substring with reversed characters (i.e., you are not getting a string from `end_index` to `start_index` , in that order). merely specifying the indexes relative to the end of the original string. Remember also that the last character of the string is -1 `end_index` argument specifies the character *after* the last character in your substring, so it's impossible to refer to the last in the original string using a negative `end_index` . Take a careful look at how we use negative indexes to extract the follo substrings:

```
var fullName = "Steven Sid Mumby";
middleName = fullName.slice(-9, -6);   // Assigns "Sid" to middleName.
firstName = fullName.slice(-16, -10);  // Assigns "Steven" to firstName.
lastName = fullName.slice(-5, -1);     // Assigns "Mumb" to lastName: not what
                                       // we want, but the best we can do with
                                       // a negative end_index.
lastName = fullName.slice(-5, 16)      // Assigns "Mumby" to lastName. Notice
                                       // we combine negative and positive ind
lastName = fullName.slice(-5)          // Also assigns "Mumby" to lastName.
```

The following code uses negative character indexes to check if the most recent letters entered into the text field `guess_txt` one of the special words in the `specialWords` array. Note that the example uses the *TextField* object's *onChanged( )* hand detect user input and the *String.toLowerCase( )* method to perform case-insensitive string comparison. For details, see *TextField.onChanged( )* and *String.toLowerCase( )* in the *Language Reference* .

```
var specialWords = ["MEGAN", "SING", "SUNDAY"];

guess_txt.onChanged = function ( ) {
  for (var i=0; i < specialWords.length; i++) {
    var guess = this.text.slice(-specialWords[i].length).toLowerCase( );
    var special = specialWords[i].toLowerCase( );
    if (guess =  = special) {
      trace("The user entered a special word!");
      break;
    }
  }
}
```

4.6.6.4 The split( ) function

So far, the string-extraction functions we've seen have retrieved only one character sequence at a time. If we want to rip out a substrings in one fell swoop, we can use the powerful *split( )* function. (Because the *split( )* function uses arrays, you may wa this function for now and come back after you've read Chapter 11.)

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

The *split( )* function breaks a string into a series of substrings and returns those substrings in an array. The *split( )* function tal following form:

```
string.split(delimiter)
```

where `string` is any literal string value or an identifier that contains a string, and `delimiter` is the character or charac indicate where `string` should be split. Commas, spaces, and tabs are typical delimiters. To break up a string at each comn example, we use:

```
theString.split(",")
```

One of the neat tricks we can pull with *split( )* is to break a sentence up into individual words. In our coverage of the *substrin substr( )* , and *slice( )* functions, we had to grab each name from the string "Steven Sid Mumby" manually. Look how much e are when we use *split( )* with a space (" ") as the *delimiter* :

```
var fullName = "Steven Sid Mumby";
var names = fullName.split(" ");  // Man, that's easy!
// Now assign the names in our array to individual variables
firstName  = names[0];
middleName = names[1];
lastName   = names[2];
```

4.6.6.5 Flash 5 string extraction performance issues

Flash 5's *substr( )* and *slice( )* functions are actually implemented as veneer atop the old Flash 4 *substring( )* function and, the take marginally longer to execute. The speed difference is on the order of milliseconds but can be noticeable in intensive strin processing. When carrying out highly repetitive operations in Flash 5, use Flash 4's *substring( )* function for optimal perform follows:

```
fullName = "Steven Sid Mumby";
// Assign "Sid" to middleName using Flash 5 substr( ) function
middleName = fullName.substr(7, 3);
// Assign "Sid" to middleName using Flash 4 substring( ) function.
// Note that character indexes start at 1 with Flash 4's substring( ).
middleName = substring(fullname, 8, 3);
```

For the benefit of Flash 5 developers, Branden Hall has rewritten the methods of the *String* class to improve their speed. See

http://chattyfig.figleaf.com/~bhall/code/string.as

In Flash Player 6, string performance was greatly improved, so this workaround is not necessary.

4.6.7 Combining String Examination with Substring Extraction

We've seen how to search for characters in a string and how to extract characters from a string. These two operations are mos when we put them together.

Most of the examples we've seen so far use literal expressions as arguments, like this:

```
var msg = "Welcome to my website!";
var firstWord = msg.substring(0, 7);  // 0 and 7 are numeric literals
```

That's a decent demonstration of the way *substring( )* works, but it doesn't represent the typical real-world use of *substring( )* .

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

often, we don't know the content of the string in advance and we must generate our arguments dynamically. For example, inst saying something static like, "Get me the substring from index 0 to index 7," we usually say something dynamic like, "Get me substring starting from the first character and ending at the first occurrence of a space in this string." This more flexible appro doesn't require us to know the content of a string in advance. Here we extract the first word of the variable `msg` , by combini *substring( )* with *indexOf( )* :

```
var firstWord = msg.substring(0, msg.indexOf(" "));
```

The expression *msg.indexOf(" ")* evaluates to the numeric index of the first space in `msg` . Our technique will work regardles space's location. This allows us to work with strings that change while our program is running and saves us from counting cha which is prone to error.

The combinations of string examination and string extraction are practically endless. Example 4-3 extracts the second word o variable without hard-coding the character indexes. In natural language, we want to "extract a substring from `msg` , starting v character after the first occurrence of a space and ending with the character before the second occurrence of a space." We stor location of the first and second spaces as variables, making the code easier to follow.

Example 4-3. Retrieving the second word of a string

```
var msg = "Welcome to my website!";
firstSpace = msg.indexOf(" ");                    // Find the first space
secondSpace = msg.indexOf(" ", firstSpace + 1);  // Find the next space
// Now extract the second word
var secondWord = msg.substring(firstSpace + 1, secondSpace);
```

4.6.8 Character Case Conversion

We can convert a string to upper- or lowercase using the built-in *toUpperCase( )* and *toLowerCase( )* functions. These functio typically used to display a string with nice formatting or to compare strings with different cases.

4.6.8.1 The toUpperCase( ) function

The *toUpperCase( )* function converts all of the characters in a string to uppercase (i.e., capital letters) and returns the convert If no uppercase version of a given character exists, the character is returned unchanged. The general form of *toUpperCase( )* i

```
string.toUpperCase()
```

where *`string`* is any literal string value or an identifier that contains a string. Here are some examples:

```
"listen to me".toUpperCase();       // Yields the string "LISTEN TO ME"
var msg1 = "Your Final Score: 234";
var msg2 = msg1.toUpperCase();      // Set msg2 to "YOUR FINAL SCORE: 234"
```

Note that *toUpperCase( )* does not affect the string on which it's called; it merely returns an uppercase copy of that string. The example shows the difference:

```
var msg = "Forgive me, I forgot to bring my spectacles.";
msg.toUpperCase();
trace(msg);    // Displays: "Forgive me, I forgot to bring my spectacles."
               // msg was unaffected by the toUpperCase( ) invocation.
```

4.6.8.2 The toLowerCase( ) function

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

The *toLowerCase( )* function changes the characters in a string from upper- to lowercase. For example:

```
// Set normal to "this sentence has mixed caps!"
normal = "ThiS SenTencE Has MixED CaPs!".toLowerCase();
```

To compare two strings in a case-insensitive manner, first convert them both to the same case, such as:

```
if (userEntry.toLowerCase( ) =  = password.toLowerCase()) {
  // They get secret access
}
```

We can use *toUpperCase( )* and *toLowerCase( )* together to capitalize the first letter of a word, as follows:

```
var word = "fLASH";
var firstLetter = word.charAt(0).toUpperCase( );
var remainderOfWord = word.substring(1).toLowerCase( );
initialCapWord = firstLetter + remainderOfWord;
trace(initialCapWord);  // Displays: Flash
```

Example 4-4 shows how we can have a little fun using the case-conversion and string functions to animate text in a text field. new method, *caseAni( )* , to all text fields. Attach the script to frame 1 of a new movie and see what happens. Note that the co several advanced techniques that we haven't encountered yet. Come back and study it again once you've read Chapter 9 and C

Example 4-4. Character case animation

```
// Add a new caseAni( ) method to the built-in TextField class.
TextField.prototype.caseAni = function (msg) {
  // Store a reference to the current text field in tf. We'll need to
  // access it from the nested function moveUppercaseLetter( ).
  var tf = this;
  tf.upperIndex = 0;  // The current location of the uppercase letter.
  tf.msg = msg;       // The message to animate.

  // Start the animation running by calling
  // the moveUppercaseLetter( ) function every 100 milliseconds.
  var id = setInterval(moveUppercaseLetter, 100);

  // Define the nested moveUppercaseLetter( ) function. Here's where we do
  // all our string manipulation.
  function moveUppercaseLetter ( ) {
    // Chop the msg string into three pieces and make the middle one uppercase
    var part1 = tf.msg.slice(0, tf.upperIndex);
    var part2 = tf.msg.charAt(tf.upperIndex);
    var part2 = part2.toUpperCase( );
    var part3 = tf.msg.slice(tf.upperIndex+1, tf.msg.length);
    tf.msg  = part1 + part2 + part3;

    // Display the formatted msg string.
    tf.text = tf.msg;

    // Reset the string for next time.
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

```
    tf.msg  = tf.msg.toLowerCase( );

    // Move the uppercase letter over one to the right.
    tf.upperIndex++;

    // If the uppercase letter is at the end of the string, move
    // it back to the beginning.
    if (tf.upperIndex > (tf.msg.length - 1)) {
      tf.upperIndex = 0;
    }
  }
}

// Try it out on a text field instance!
this.createTextField("msgOutput_txt", 1, 0, 0, 300, 30);
// Use a monospaced font...
msgOutput_txt.setNewTextFormat(new TextFormat("_typewriter", 16));
msgOutput_txt.caseAni("my what fat cheeks you have");
```

4.6.9 Character Code Functions

In Section 4.5.2.3, we saw how to insert characters into a string using escape sequences. ActionScript also includes two built-functions for working with character code points in strings: *fromCharCode( )* and *charCodeAt( )* .

4.6.9.1 The fromCharCode( ) function

We can create any character or series of characters by invoking the *fromCharCode( )* function. Unlike the other string functio *fromCharCode( )* is not called on a string literal or an identifier that contains a string; it is called as a method of the built-in *St* like this:

```
String.fromCharCode(code_ point1, code_ point2, ...)
```

Every *fromCharCode( )* call starts with *String.fromCharCode* . Then one or more code points (representing the characters we create) are supplied as arguments. Unlike Unicode escape sequences, which must be in hex, the code points in a *fromCharCo* can be expressed in either decimal or hex. If you're unfamiliar with hex numbers you may, therefore, find *fromCharCode( )* ea than Unicode escape sequences. For values less than 128, *fromCharCode( )* essentially converts a numeric code to its equival character. Here are some examples:

```
// Set lastName to "moock"
lastName = String.fromCharCode(109, 111, 111, 99, 107);
// For comparison, let's do the same thing with Unicode escape sequences
lastName = "\u006D\u006F\u006F\u0063\u006B";
// Make a copyright symbol
copyNotice = String.fromCharCode(169) + " 2003";
// Make a euro sign
price = "5.99 " + String.fromCharCode(0x20AC);
```

You may encounter the deprecated Flash 4 character-creation functions, *chr( )* and *mbchr( )* , when examining legacy code. T *fromCharCode( )* function should be used in Flash 5 and later.

4.6.9.2 The charCodeAt( ) function

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

To determine the code point of any character in a string, we use the *charCodeAt( )* function, which takes the following form:

```
string.charCodeAt(index)
```

where `string` is any literal string value or an identifier that contains a string, and `index` is the position of the character examining. The *charCodeAt( )* function returns a decimal integer that matches the Unicode code point of the character at `ind` values less than 128, *charCodeAt( )* essentially converts a character to its equivalent ASCII code. For example:

```
var msg = "A is the first letter of the Latin alphabet.";
trace(msg.charCodeAt(0));  // Displays: 65 (the code point for "A")
trace(msg.charCodeAt(1));  // Displays: 32 (the code point for a space)
```

We normally use *charCodeAt( )* to perform string handling with characters we can't type directly using a keyboard. For exam following code we check whether a character is the copyright symbol:

```
msg = String.fromCharCode(169) + " 2003";
if (msg.charCodeAt(0) =  = 169) {
  trace("The first character of msg is a copyright symbol.");
}
```

You may encounter the deprecated Flash 4 code point functions, *ord( )* and *mbord( )* , when examining legacy code. The *cha* function should be used in Flash 5 and later.

4.6.10 Executing Code in a String with eval

In ActionScript, the *eval( )* function converts a string to an identifier (i.e., a variable name). But to thoroughly understand the ActionScript *eval( )* function, you must first understand how JavaScript's analogous *eval( )* function works. In JavaScript, *eva* built-in function that converts any string to a block of code and then executes that block of code. The syntax for JavaScript's *e*

```
eval(string)
```

When *eval( )* is executed in JavaScript, the interpreter converts `string` to executable code, runs that code, and returns the value (if a value is generated). Consider the following JavaScript examples:

```
eval("parseInt('1.5')");  // Calls the parseInt( ) function, which returns 1
eval("var x = 5");        // Creates a new variable named x and sets its value
```

If you've never seen *eval( )* before, you may be thinking, "When would I ever have a string with code in it? Why not just write out?" Answer: because *eval( )* lets you dynamically generate code when you need to. For example, *eval( )* could be used to programmatically create functions to be assigned as event handler callbacks for a series of similar objects.

ActionScript's *eval( )* function supports a small subset of its JavaScript cousin's functionality: it works only when its argumen identifier. Hence, ActionScript's *eval( )* function can only retrieve the data associated with the specified identifier. For examp

```
var num = 1;
var person1 = "Eugene";
trace (eval("person" + num));  // Displays: "Eugene"
```

Even in this pared-back form, *eval( )* is quite useful. Here we generate a series of movie clips dynamically with a loop. We pl clips in an array by using *eval( )* to refer to them:

```
for (var i = 0; i < 10; i++) {
  duplicateMovieClip("ballParent", "ball" + i, i);
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

```
  balls[i] = eval("ball" + i);
}
```

Note, however, that in nearly all cases *eval( )* can be replaced by the array-access operator to generate dynamic clip reference example:

```
duplicateMovieClip("ballParent", "ball" + i , i);
balls[ballCount] = _root ["ball" + i];
```

The *eval( )* function was used frequently in Flash 4 to generate variables dynamically because support for arrays wasn't availa Flash 5. Such uses of *eval( )* should be rewritten using arrays when upgrading legacy code.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Chapter 4. Primitive Datatypes

4.7 The Boolean Type

Boolean data is used to represent the logical states of truth and falsehood. Hence there are only two legal values of the *boolean* datatype: `true` and `false`. Notice that there are no quotation marks around the words `true` and `false` because Boolean data is *not* string data. The keywords `true` and `false` are reserved primitive data values and cannot be used as variable names or identifiers.

We use Boolean values with conditional expressions to add logic to the execution of code. For example, we might assign the value `true` to a variable that tracks the status of a spaceship's firepower:

```
shipHasDoubleShots = true;
```

By comparing `shipHasDoubleShots` to the Boolean literal `true`, we can decide how much damage to inflict when a shot hits its target:

```
if (shipHasDoubleShots =  = true) {
  // Shoot them with twice the power.
  // This will be reached if the comparison is true.
} else {
  // Shoot them with a single dose.
  // This will be reached if the comparison is false.
}
```

When the double-shot power runs out, we can set the variable to `false`:

```
shipHasDoubleShots = false;
```

This will cause the larger expression *shipHasDoubleShots = = true* to become `false`, causing the single-damage-dose script to execute when a shot hits its target.

All comparison operators express results with Boolean values. When we ask, "Is the user's guess the same as the password?" the answer is given as a Boolean:

```
// userGuess =  = password will yield either true or false
if (userGuess =  = password) {
  gotoAndStop("secretContent");
}
```

And when we ask, "Is the movie clip rotated more than 90 degrees?" the answer, again, is a Boolean:

```
// theClip_mc._rotation > 90 will yield either true or false
if (theClip_mc._rotation > 90) {
  // Fade out the clip if it's rotated past 90 degrees
  theClip_mc._alpha = 50;
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

```
}
```

Many internal ActionScript properties and methods describe the Flash movie environment in Boolean terms. For example, if we ask, "Is the spacebar being pressed?" the interpreter answers with a Boolean: `true` (yes) or `false` (no):

```
// Key.isDown( ) is a function that returns either true or false
if (Key.isDown(Key.SPACE)) {
  // Spacebar is being pressed, so make our spaceship fire
}
```

In Chapter 5, we'll learn how to phrase complex logical expressions using Boolean operators.

4.7.1 Using Boolean Values to Build a Preloader

Let's consider an applied Boolean example. Suppose we have a document with 500 frames and lots of content. The beginning of our opening sequence, frame 20, is labeled `intro`. We put the following code on frame 2 of that movie's main timeline:

```
if (this._framesloaded >= this._totalframes) {
  this.gotoAndPlay("intro");
} else {
  this.gotoAndPlay(1);
}
```

When the movie plays, the playhead passes frame 1 and enters frame 2. The ActionScript interpreter reaches the conditional statement and evaluates the Boolean expression within it: *this._framesloaded >= this._totalframes*. While the movie is still loading, `this._framesloaded` is less than the total number of frames in our movie (`this._totalframes`). If `this._framesloaded` is not greater than or equal to `this._totalframes`, then the expression *this._framesloaded >= this._totalframes*, yields `false`. Therefore, the statement *this.gotoAndPlay("intro")* is skipped and the statement *this.gotoAndPlay(1)* is executed instead. The *this.gotoAndPlay(1)* statement sends the playhead back to frame 1 and plays the movie. When the playhead enters frame 2, our code is executed again. The playhead keeps looping in this way until the expression *this._framesloaded >= this._totalframes* yields the value `true` (i.e., until all the frames have loaded). At that point, the statement *this.gotoAndPlay("intro")*, which sends the playhead to the label `intro`, is executed. There we can safely start our movie, now that all of the frames have loaded.

Whammo! You've created a preloader based on a Boolean expression. This is solid stuff. We'll learn much more about conditionals and controlling movies with Booleans in Chapter 7.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Chapter 4. Primitive Datatypes

4.8 Undefined

Most of the datatypes we've explored so far have been used for storing and manipulating information. The *undefined* datatype has a more narrow purpose: it is used to check whether a variable exists or whether a variable has been assigned a value yet. The *undefined* datatype has only one legal value, the primitive value `undefined`.

When we first define a variable, it is assigned the value `undefined` by default:

```
var velocity;
```

To the interpreter, the preceding statement reads:

```
var velocity = undefined;
```

To check whether a variable has a value, we can compare the variable to `undefined`, as in:

```
if (myVariable !=  = undefined) {
  // myVariable has a value, so proceed as desired...
}
```

To remove a variable or object property, you should use the *delete* operator rather than assigning the value `undefined` to the variable or property. Assigning `undefined` to a property leaves the property intact but merely sets its value to `undefined`. Deleting the property removes it entirely.

Note that, in ActionScript, an `undefined` value is converted to the empty string when used as a string. For example, in the following code we create two variables, `firstName` and `lastName`, which we concatenate together and assign to the variable `fullName`. The variable `firstName` has no value (the value is `undefined`), so it converts to `""`.

```
var firstName;
var lastName = "Moock";
var fullName = firstName + lastName;  // Sets fullName to "Moock"
```

In JavaScript, the `undefined` value is converted to the string "undefined" when used in a string context, so the above code in JavaScript would assign the value "undefinedMoock" to `fullName`. ActionScript converts `undefined` to `""` for the sake of backward compatibility. However, in Flash MX and later, when an `undefined` value is sent to the Output window via the *trace( )* function, the word "undefined" appears (this makes `undefined` easier to detect during debugging).

Because there was no *undefined* type in Flash 4 ActionScript, many Flash 4 programs used the empty string to check whether a variable had a useful value. Code like this was common in Flash 4:

```
if (myVar eq "") {
  // Don't do anything yet: myVar is undefined
}
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

If Flash converted `undefined` to anything other than `""` in a string context, old code such as this wouldn't work in Flash Player 5 and later.

Note that ActionScript returns `undefined` for both variables that do not exist and variables that have been declared but have no value. This is also a departure from JavaScript, where references to variables that do not exist cause an error.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Chapter 4. Primitive Datatypes

4.9 Null

Intellectually, the *null* type is nearly identical to the *undefined* type. Like the *undefined* datatype, the *null* datatype is used to represent a lack of data and has only one legal value, the primitive value `null`. However, the `null` value is not assigned by the interpreter automatically; it must be assigned by us deliberately.

We assign `null` to a variable, array element, or object property to indicate that the specified data container does not contain a legal number, string, Boolean, array, or object value. For example, we might assign an initial value of `null` to an object property to indicate that it exists but has not yet been assigned a useful value.

To remove a variable or object property, you should use the *delete* operator rather than assigning the value `null` to the variable or property.

Note that `null` compares equal only to itself and `undefined`:

```
null =  = undefined;   // true
null =  = null;        // true
```

To guarantee that a variable or property contains `null` (rather than `undefined`), check its datatype using the *typeof* operator:

```
var x = null;
if (typeof x =  = "null") {
  trace("x is null");
}
```

Alternatively, as of Flash Player 6, we can use the strict equality operator (`= = =`), as follows:

```
if (x =  =  = null) {
  trace("x is null");
}
```

Though `null` and `undefined` are similar, their roles in ActionScript programs are distinct. The `null` value is used by programmers to indicate that a variable or property has been left undetermined intentionally. By contrast, the `undefined` value is assigned by the interpreter automatically whenever no other value is available. Consider a photo album application that displays a series of images sequentially. The application is initialized with `title` and `description` variables. If we choose not to bother with a description for a particular album, we set `description` to `null`.

```
var title = "My Holiday Photos";
var description = null;
```

The ensuing initialization code can then tell that `description` was omitted intentionally.

```
// If there's a valid description...
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

```
if (description !=  = null) {
  // ...create description text field.
}
```

If we use `undefined` instead of `null`:

```
var title = "My Holiday Photos";
var description = undefined;
```

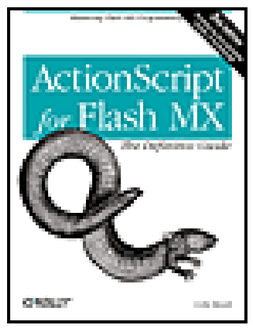
the initialization code has no way to tell whether the interpreter set `description`'s value automatically or we set it intentionally. The distinction is subtle, but it is valuable in large programs. Here we expand our earlier initialization code to display an error message when no description value has been set.

```
// If there's a valid description...
if (description =  =  = undefined) {
  trace("WARNING: No description value was specified!");
} else if (description !=  =  = null) {
  // ...create description text field.
}
```

By distinguishing between `undefined` and `null` in our initialization code, we can provide accurate debugging feedback and catch potential programmer oversight. For another example of real-world `null` usage, see the *TextFormat* class's constructor parameters in the *Language Reference*.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir



ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Chapter 4. Primitive Datatypes

4.10 Onward!

You've learned a lot more so far than you may have realized. We've built a solid foundation for the advanced topics coming in later chapters. If you haven't mastered all the details yet, don't worry. Your training will come back to you when you need it. Keep the faith during the next chapter, in which you'll learn how to merge and transform data. Following that, we'll get into some more fully formed applied examples.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Part I: ActionScript Fundamentals

Chapter 5. Operators

An *operator* is a symbol or keyword that manipulates, combines, or transforms data. If you're new to programming, you'll notice that some mathematical operators, such as + (addition) and - (subtraction), are very familiar. In other cases, you'll have to learn special programming syntax, even if the concepts are familiar. For example, to multiply two numbers, ActionScript uses the symbol * (the multiplication operator) instead of the x typically taught in grade school. For example, this multiplies 5 times 6:

```
5 * 6;
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Chapter 5. Operators

5.1 General Features of Operators

Though each operator has its own specialized task, all operators share a number of general characteristics. Before we consider operators individually, let's see how they behave generally.

5.1.1 Operators and Expressions

Operators perform actions using the data values (*operands*) supplied. For example, in the operation `5 * 6`, the numbers 5 and 6 are the *operands* of the multiplication operator (*). The operands can be any kind of expression; for example:

```
total = player1score + bonusScore;              // Operands are variables
answer = (x+y) - (Math.PI * radius * radius);    // Operands are complex expre
```

Observe in the second example that both the left and right operands of the - operator are expressions that themselves involve other operations. We can use complex expressions to create even larger expressions, such as:

```
answer =((x+y) - (Math.PI * radius * radius)) / 2  // Divide the whole thing b
```

When expressions become very large, consider using variables to hold interim results for both convenience and clarity. Remember to name your variables descriptively, such as:

```
var radius = 10;
var height = 25;
var circleArea = (Math.PI * radius * radius);
var cylinderVolume = circleArea * height;
```

5.1.2 Number of Operands

Operators are sometimes categorized according to how many operands they take (i.e., require or operate on). Some ActionScript operators take one operand, some take two, and one even takes three:

```
-x                                                  // One operand
x * y                                               // Two operands
(x =  = y) ? "true result" : "false result"  // Three operands
```

Single-operand operators are called *unary* operators; operators that take two operands are called *binary* operators; operators that take three operands are called *ternary* operators. For our purposes, we'll look at operators according to what they do, not the number of operands they take.

5.1.3 Operator Precedence

Operators' *precedence* determines which operation is performed first in an expression with multiple operators. For example, when multiplication and addition occur in the same expression, multiplication is performed first:

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

```
4 + 5 * 6  // Yields 34, because 4 + 30 = 34
```

The expression `4 + 5 * 6` is evaluated as `4 + (5 * 6)` because the * operator has higher precedence than the `+` operator. Similarly, when less-than (`<`) and concatenation (`+`) operators occur in the same expression, concatenation occurs first. For e this statement displays `false` in the Output window (which is a surprise if you don't know the precedence of the `<` and `+` op

```
trace("result: " + "a" < "b");
```

First, "result: " is joined with "a", creating the new string "result: a". This new string is then compared with "b", which yields . Parentheses should be added to force the intended order of operation, as follows:

```
trace("result: " + ("a" < "b"));  // Displays: result: true
```

When in doubt, or to ensure a different order of operation, use parentheses, which have the highest precedence:

```
(4 + 5) * 6  // Yields 54, because 9 * 6 = 54
```

Even if not strictly necessary, parentheses can make a complicated expression more readable. The expression:

```
// x is greater than y, or y equals z
x > y || y =  = z
```

may be difficult to comprehend without consulting a precedence table. It's a lot easier to read with parentheses added:

```
(x > y) || (y =  = z)  // Much better!
```

Table 5-1 shows the precedence of each operator. Operators with the highest precedence (at the top of the table) are executed Operators with the same precedence are performed in the order they appear in the expression, usually from left to right, unless associativity is right to left (see the Associativity column in Table 5-1). See Section 5.1.4 for details.

Table 5-1. ActionScript operator associativity and precedence

Operator	Precedence	Associativity	Description
.	15	left to right	object property access
`[ ]`	15	left to right	array element access
`( )`	15	left to right	parentheses
function `( )`	15	left to right	function call
`x++`	14	left to right	postfix increment
`x--`	14	left to right	postfix decrement
`++x`	14	right to left	prefix increment
`--x`	14	right to left	prefix decrement
`-`	14	right to left	unary negation
`~`	14	right to left	bitwise *NOT*
`!`	14	right to left	logical *NOT*
`new`	14	right to left	create object/array
`delete`	14	right to left	remove object/property/array e
`typeof`	14	right to left	determine datatype

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

`void`	14	right to left	return undefined value
`*`	13	left to right	multiply
`/`	13	left to right	divide
`%`	13	left to right	modulo division
`+`	12	left to right	addition or string concatenatic
`-`	12	left to right	subtraction
`<<`	11	left to right	bitwise left shift
`>>`	11	left to right	bitwise signed right shift
`>>>`	11	left to right	bitwise unsigned right shift
`<`	10	left to right	less than
`<=`	10	left to right	less than or equal to
`>`	10	left to right	greater than
`>=`	10	left to right	greater than or equal to
`instanceof`	10	left to right	check an object's class
`= =`	9	left to right	equality
`!=`	9	left to right	not equal to
`= = =`	9	left to right	strict equality
`!= =`	9	left to right	strict inequality
`&`	8	left to right	bitwise *AND*
`^`	7	left to right	bitwise *XOR*
`|`	6	left to right	bitwise *OR*
`&&`	5	left to right	logical *AND*
`||`	4	left to right	logical *OR*
`? :`	3	right to left	conditional
`=`	2	right to left	assignment
`+=`	2	right to left	add and reassign
`-=`	2	right to left	subtract and reassign
`*=`	2	right to left	multiply and reassign
`/=`	2	right to left	divide and reassign
`%=`	2	right to left	modulo division and reassign
`<<=`	2	right to left	bit-shift left and reassign
`>>=`	2	right to left	bit-shift right and reassign
`>>>=`	2	right to left	bit-shift right (unsigned) and r
`&=`	2	right to left	bitwise *AND* and reassign
`^=`	2	right to left	bitwise *XOR* and reassign
`|=`	2	right to left	bitwise *OR* and reassign
,	1	left to right	comma

5.1.4 Operator Associativity

As we've just seen, operator precedence indicates the pecking order of operators: those with a higher precedence are executed those with a lower precedence. But what happens when multiple operators occur together and have the same level of preceder

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

such a case, we apply the rules of *operator associativity* , which indicate the direction of an operation. Operators are either lef associative (performed left to right) or right-associative (performed right to left). For example, consider this expression:

```
a = b * c / d
```

The * and / operators are left-associative, so the * operation on the left (`b * c`) is performed first. The preceding example is equivalent to:

```
a = (b * c) / d
```

In contrast, the = (assignment) operator is right-associative, so the expression:

```
a = b = c = d
```

says, "assign `d` to `c` , then assign `c` to `b` , then assign `b` to `a` ," as in:

```
a = (b = (c = d))
```

Operator associativity is fairly intuitive, but if you're getting an unexpected value from a complex expression, consult Table 5 add extra parentheses. We'll note cases in which associativity is a common source of errors throughout the remainder of this c

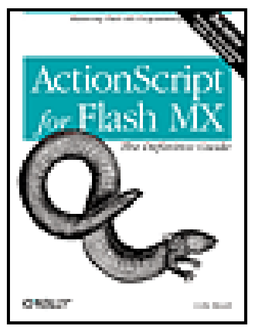
5.1.5 Datatypes and Operators

Some operators accept varying datatypes as operands. Depending on the datatype of an operand, the effect of an operator may The + operator, for example, performs addition when used with numeric operands, but it performs concatenation when used v string operands. If operands are of different datatypes or of the wrong type, ActionScript will perform type conversion accord the rules described in Chapter 3, which can have serious effects on your code.

The remainder of this chapter offers reference-style descriptions of each ActionScript operator. Newer programmers may wis skim past topics that have not yet been covered in detail.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir



ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Chapter 5. Operators

5.2 The Assignment Operator

We've already used the assignment operator frequently. It can place a value into a variable, array element, or object property. Assignment operations take the form:

```
identifier = expression
```

The *identifier* is the variable, array element, or object property into which we want to place our value. The *expression* represents the value (i.e., data) that we want to store. For example:

```
x = 4;                      // Assign 4 to the variable x
x = y;                      // Assign the value of y to the variable x
name = "dj duong";          // Assign a string to the variable name
products[3] = "Flash";      // Assign a string to the 4th element of products
                            // (array indices are zero-relative)
// Assign a number to the area property of square
square.area = square.width * 2;
```

We can also perform multiple assignment operations at once, like this:

```
x = y = 4;  // Set both x and y to 4
```

Remember that assignment operations have right-to-left associativity, so 4 is assigned first to `y`, and then the value of `y` (which is now 4) is assigned to `x`.

The left side of an assignment operation can never involve an operation. For example, this code is common in mathematics, but it is illegal in programming:

```
x + y = 10;  // ERROR!
```

As we'll see later, equality comparisons between two expressions are performed with the `= =` and `!=` operators. For example, the following checks whether `x + y` is equal to 10:

```
var x = 1;
var y = 8;
trace(x + y =  = 10);  // Displays: false
```

Again, this is not quite the same as mathematics, where $x + y = 10$ asserts that $x + y$ is equal to 10, rather than checking whether that is the case.

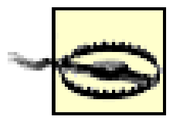
5.2.1 Combining Operations with Assignment

Assignment operations are often used to set a variable's new value, based in part on its old value. For example:

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

```
counter = counter + 10;                 // Add 10 to the current value of counter
xPosition = xPosition + xVelocity;  // Add xVelocity to xPosition
score = score / 2;                          // Divide score by two
```



Don't confuse an equals sign, which is used to assign a value to a variable, with the algebraic equals sign. For an explanation of the difference, see Section 2.2 in Chapter 2.

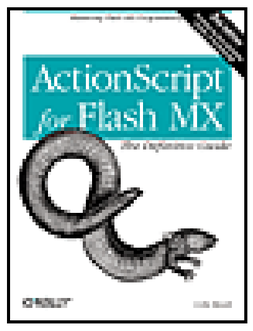
ActionScript supports a shorthand version of assignment, called *compound assignment*,that combines operators such as +, -, and / with the assignment operator to form a single "calculate-while-assigning" operation. So, to combine addition with assignment we use +=. To combine division with assignment, we use /=. Hence, the previous examples can be written more succinctly with compound assignment operators as follows:

```
counter += 10;
xPosition += xVelocity;
score /= 2;
```

Refer to Table 5-1 for a list of the compound assignment operators.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir



ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Chapter 5. Operators

5.3 Arithmetic Operators

The arithmetic operators perform mathematical operations on numeric operands. If you use nonnumeric operands with most of the arithmetic operators, ActionScript attempts to convert the foreign data to a number. For example, `false - true` evaluates to -1, because `false` converts to the numeric value 0 and `true` converts to 1. Similarly, the expression `"3" * "5"` results in the number 15, because the strings "3" and "5" are converted to the numbers 3 and 5 before the multiplication is performed. The `+` operator, however, presents a special case: when used with at least one string operand, it performs string concatenation, not mathematical addition.

If an attempt to convert a nonnumeric operand to a number fails, the operand is set to the special numeric value `NaN`. This results in the entire operation yielding `NaN`. Refer to Table 3-1 for details on numeric conversion.

5.3.1 Addition

The *addition* operator returns the sum of its two operands:

```
operand1 + operand2
```

In order to return a meaningful mathematical result, the operands of `+` should be expressions that yield a numeric value, such as:

```
234 + 5           // Returns 239
(2 * 3 * 4) + 5   // Returns 29
```

The addition operator is unique among the arithmetic operators in that if one or both of its operands are strings, it performs as a string concatenation. Refer to Section 4.6.1 in Chapter 4.

5.3.2 Increment

A handy variation on addition, the *increment* operator accepts a single operand and simply adds 1 to its current value. Increment has two general forms, called *prefix increment* and *postfix increment*, as follows:

```
++operand    // Prefix increment
operand++    // Postfix increment
```

In both forms, increment adds 1 to a variable, array element, or object property, such as:

```
var x = 1;
x = x + 1;  // x is now 2 (the verbose syntax)
++x;        // Add 1 using the prefix operator: x is now 3
x++;        // Add 1 using the postfix operator: x is now 4
```

When used in isolation, there is no difference between prefix and postfix increment, although postfix increment is

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

more common by convention.

However, when used in larger expressions, prefix and postfix increment have different behaviors: prefix increment adds 1 to *operand* and returns the value of *operand* + 1; postfix increment adds 1 to *operand* but returns the value of *operand* itself, not *operand* + 1:

```
var x = 1;
// Prefix increment: x is incremented first, so y is set to 2
var y = ++x;

var x = 1;
// Postfix increment: y is set to 1, then x is incremented to 2
var y = x++;
```

We'll revisit the increment operators in Chapter 8.

5.3.3 Subtraction

The *subtraction* operator subtracts the second operand from the first operand. It takes the general form:

```
operand1 - operand2
```

The operands can be any valid expression. If either operand is not of the *number* type, and conversion to a number fails, the operation yields NaN:

```
234 - 5  // Yields 229
5 - 234  // Yields -229
```

To determine the absolute (i.e., positive) difference between two numbers, see the *Math.abs( )* method in the ActionScript Language Reference.

5.3.4 Decrement

The *decrement* operator is analogous to the increment operator, but it subtracts 1 from its operand's current value instead of adding 1. Like increment, decrement has two general forms, called *prefix decrement* and *postfix decrement*, as follows:

```
--operand  // Prefix decrement
operand--  // Postfix decrement
```

In both forms, it is used to subtract 1 from a variable, array element, or object property. Prefix decrement subtracts 1 from *operand* and returns the value of *operand* - 1; postfix decrement subtracts 1 from *operand* but returns the value of *operand* itself, not *operand* - 1. As with the increment operators, the form of decrement used matters only if the operand is part of a larger expression:

```
var x = 10;
var y;
x = x - 1;    // x is now 9
--x;          // x is now 8 (prefix decrement)
x--;          // x is now 7 (postfix decrement)
y = --x;      // y is now 6, x is now 6 (prefix decrement)
y = x--;      // y is still 6, x is now 5 (postfix decrement)
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

5.3.5 Multiplication

The *multiplication* operator multiplies two numeric operands and returns the result (i.e., the *product*). Multiplication takes the general form:

```
operand1 * operand2
```

The operands can be any valid expression. The * symbol used for multiplication is in lieu of the x ("times") symbol used in traditional mathematics. If either operand is not of the *number* type, and conversion to a number fails, the operation yields `NaN`:

```
6 * 5  // Returns 30
```

5.3.6 Division

The *division* operator divides the first operand (the *numerator*) by the second operand (the *divisor*) and returns the result (the *quotient*). Division takes the general form:

```
operand1 / operand2
```

The operands must be valid numeric expressions. The `/` symbol used for division is in lieu of the ÷ symbol used in traditional mathematics. If either operand is not of the *number* type, and conversion to a number fails, the operation yields `NaN`. If it is necessary to express a fractional result, the quotient is a floating-point number, even if both operands are integers:

```
20 / 5  // Returns 4
5 / 4   // Returns 1.25; In other languages, the result may be 1, not 1.25
```

Note that some other languages, such as Director's Lingo language, return an integer unless at least one operand is a float.

If the divisor (denominator) is zero:

- The result is `NaN` if the numerator is zero or nonnumeric
- The result is `Infinity` if the numerator is a positive number
- The result is `-Infinity` if the numerator is a negative number

Here are some examples:

```
trace (0/0);   // NaN
trace ("a"/0); // NaN
trace (1/0);   // Infinity
trace (-1/0);  // -Infinity
```

If there is any possibility of the divisor being zero, check its value before performing the division, such as:

```
if (numItems != 0) {
  trace ("Average is" + total / numItems);
} else {
  trace ("There are no items for which to calculate the average");
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

```
}
```

Note that in some languages, attempting to divide by zero causes an error.

5.3.7 Modulo Division

The *modulo* operator performs so-called *modulo division.* It returns the remainder (i.e., *modulus*) that results when the first operand is divided by the second. Modulo division takes the general form:

```
operand1 % operand2
```

For example, `14 % 4` returns the value 2 because 4 divides evenly into 14 three times, with 2 being the remainder.

The operands of the modulo operator can be any valid numeric expression, including integers and (unlike in C and C++) floating-point numbers. For example, `5 % 4` is 1, and `5 % 4.5` is 0.5. If either operand is not of the *number* type, and conversion to a number fails, the operation yields `NaN`.

If a number is even, the modulo will be zero when we divide the number by two. We can use the trick shown in Example 5-1 to test whether a number is even or odd.

Example 5-1. Using modulo division to test for even numbers

```
var x = 3;
if (x%2 =  = 0) {
  trace("x is even");
} else {
  trace("x is odd");
}
```

5.3.8 Unary Negation

The *unary negation* operator takes only one operand. It switches the operand's sign (i.e., positive becomes negative, and negative becomes positive). Unary negation takes the general form:

```
-operand
```

The operand can be any valid expression. Here we test whether something's horizontal position is greater than the positive limit or less than the negative limit:

```
if (xPos > xBoundary || xPos < -xBoundary) {
  // We've gone too far
}
```

The unary negation operator is simply a negative sign (-). It is equivalent to the subtraction operator, where the first operand is 0, such as:

```
0 -operand
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Chapter 5. Operators

5.4 The Equality and Inequality Operators

We use the *equality* operator (`= =`) to test whether two expressions have the same value. An equality test takes the general form:

```
operand1 = = operand2
```

where *`operand1`* and *`operand2`* can be any valid expression. The equality operator can compare operands of any type. When *`operand1`* and *`operand2`* are equal, the expression returns the Boolean value `true`; when they differ, it returns the Boolean value `false`. For example:

```
// Store the value 2 in x.
var x = 2;
// Is x equal to 1?
if (x = = 1) {
  // false; nothing is displayed
  trace("x is equal to 1);
}
// Is x equal to 2?
if (x = = 2) {
  // true; message is displayed
  trace("x is equal to 2);
}
```

> The equality operator is created using two equals signs in a row (`= =`). It determines whether two expressions are equal, and it should not be confused with the assignment operator (`=`), which is a single equals sign used to assign a new value to a variable. Flash MX also supports the strict equality operator, created using three equals signs (`= = =`). See the Section 5.5 later in this chapter.

Consider this example:

```
if (x = 5) {
  trace("x is equal to 5")
}
```

This example does *not* check whether `x` equals 5. Instead, it *sets* `x` equal to 5. It then evaluates the variable `x` as a Boolean expression, which yields `true` (positive numbers convert to the Boolean value `true`). Here is what our example looks like to the interpreter:

```
x = 5;
if (x) {
  trace("x is equal to 5")
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

```
}
```

The proper construction is as follows:

```
// Use =  = instead of =
if (x =  = 5) {
  trace("x is equal to 5")
}
```

5.4.1 Primitive Datatype Equality

For the primitive datatypes, the result of most equality tests is fairly intuitive. Table 5-2 lists the rules that govern equality for each primitive datatype.

Table 5-2. Equality of primitive datatypes

Type	Terms of equality (both operands must be of given type)

Number	If *operand1* is the same number as *operand2*, the result is `true`. If both operands are `+Infinity` or both are `-Infinity`, the result is `true`. If both operands are either `-0` or `+0`, the result is `true`. For all other combinations, including if *either* or *both* operands are `NaN`, the result is `false`:
	`1 =  = 4 // false`
	`4 =  = 4 // true`
	`NaN =  = NaN // false`
	`+Infinity = -Infinity // false`

String[1]	Performs case-sensitive string comparison. If *operand1* and *operand2* are strings of the same length that contain the exact same sequence of characters, the result is `true`; otherwise, the result is `false`:
	`"Flash" =  = "Flash" // true`
	`"O'Reilly" =  = "O Reilly" // false`
	`"Moock" =  = "moock" // false ("m" and "M" differ)`

Boolean	If both operands are `true` or both operands are `false`, the result is `true`; otherwise, the result is `false`:
	`true =  = true // true`
	`false =  = false // true`
	`true =  = false // false`

`undefined`	If both operands are `undefined`, or if one operand is `undefined` and the other is `null`, the result is `true`; otherwise, the result is `false`.
`null`	If both operands are `null`, or if one operand is `undefined` and the other is `null`, the result is `true`; otherwise, the result is `false`.
Composite datatypes	See Section 5.4.2.

[1] Flash 4's string equality operator, eq, is deprecated. The = = operator should be used in Flash 5 and Flash MX.

5.4.2 Composite Datatype Equality

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

As shown in Section 3.6 in Chapter 3, variables containing composite data (objects, arrays, functions, or movie clips) store *references* to the data and not the data itself; hence, it is possible for two variables to refer to the same underlying item. Two such operands are considered equal if and only if they refer to the same underlying composite datum, not if the operands refer to two different items that contain identical contents. Even if two operands can be converted to the same primitive value, they are not necessarily considered equal.

The following examples illustrate how ActionScript compares the references that point to the composite data, not the data itself. In the first example, the operands (`nameList1` and `nameList2`) refer to arrays that have the same elements but are actually two distinct arrays. The references are therefore different, and the comparison evaluates to `false`:

```
nameList1 = ["Linkovich", "Harris", "Sadler"];
nameList2 = ["Linkovich", "Harris", "Sadler"];
// The two arrays are not considered equal, so
// the trace( ) statement doesn't execute.
if (nameList1 =  = nameList2) {
  trace("They're the same");
}
```

In this example, `cities` and `canadianCities` both refer to the same array, so they are equal:

```
canadianCities = ["Toronto","Montreal","Vancouver"];
cities = canadianCities;
// The two arrays are considered equal, so
// the trace( ) statement does execute.
if (cities =  = canadianCities) {
  trace("They're the same");
}
```

In this example, `myFirstBall` and `mySecondBall` have the same constructor (i.e., are both objects derived from the same class), but they exist as separate (unequal) instances:

```
function Ball ( ) {
  // ...initialization code goes here...
}
myFirstBall = new Ball( );
mySecondBall = new Ball( );
myFirstBall =  = mySecondBall  // false
```

As shown in Chapter 3, composite data values are compared by reference, not by value.

To duplicate an array's contents without copying the array reference, we can use the *Array.slice( )* method. In this example, we copy the elements from the `dishes` array into `kitchenItems`:

```
dishes = [ "cup", "plate", "spoon" ];
kitchenItems = dishes.slice(0, dishes.length);
trace(kitchenItems =  = dishes);  // Displays: false
```

Now `kitchenItems` and `dishes` each contain their own private copy of the array elements and can alter them without affecting each other.

5.4.3 Equality and Datatype Conversion

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

We've seen the results of equality tests when the two operands have the same datatype, but what happens when we compare operands of different datatypes, such as a string and a number? For example:

```
"asdf" =  = 13;
```

When the operands have disparate datatypes, the interpreter performs a type conversion before performing the comparison. Here are the rules the interpreter follows:

1. If both operands are of the same type, compare them and return the result. (If `null` is compared to `undefined`, `true` is returned.)
2. If one operand is a number and the other operand is a string, convert the string to a number and go back to step 1.
3. If one operand is a Boolean, convert the Boolean to a number (`true` = 1, `false` = 0) and go back to step 1.
4. If one operand is an object, invoke the *valueOf( )* method of the object to convert it to a primitive type. Return `false` if this is not possible. Otherwise, go back to step 1.
5. Return `false` if the previous steps don't obtain a valid result.

Note that if one operand is an object and the other is a Boolean, the Boolean will be converted to a number and compared to the primitive value of the object. This means that `someObject = = true` is normally `false`, even if `someObject` exists, because `someObject` is converted to a number or a string for the comparison, while `true` is converted to the number 1. To force `someObject` to be treated as a Boolean in a comparison, use the *Boolean( )* function, like this:

```
Boolean(someObject) =  = true    // Returns true if someObject exists,
                                 // or false if it doesn't
```

Conversions caused by equality operations favor the number type. If you're wondering about the results of the various type conversions just described, see Section 3.4 in Chapter 3. Note that type conversions performed during a comparison do not alter the original item's stored value or datatype. The results of the temporary conversion are discarded once the expression has been evaluated.

5.4.4 The Inequality Operator

The *inequality* operator (also called the *does-not-equal* or *not-equal-to* operator) returns the opposite Boolean result of the equality operator. It is often more readable to say, "If `x` is *not* equal to `y`, do this," than to say, "If `x` is equal to `y`, don't do anything, otherwise do this," as shown later in Example 5-2. An inequality test takes the general form:

```
operand1 != operand2
```

For example:

```
var a = 5;
var b = 6;
var c = 6;
a != b  // true
b != c  // false
```

The inequality operator follows the same type conversion rules as the equality operator and always yields the opposite result, including when `NaN` is used as an operand:

```
NaN != 7    // true
NaN != NaN  // true!
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

In some languages, including Flash 4 ActionScript, the <> operator is used as the inequality operator. Flash 4 used the deprecated ne operator for string comparisons. In Flash 5 and Flash MX, != should be used in lieu of <> and ne for comparisons, regardless of the operands' datatypes. See also the *logical NOT* operator (!) discussed later in this chapter.

5.4.5 Common Uses of Equality Operations

We'll frequently use equality operations to form Boolean expressions within conditional statements or to assign a Boolean value to a variable. Example 5-2 shows both situations and demonstrates the != and = = operators in action.

Example 5-2. Using the equality and inequality operators

```
version = getVersion( );      // Retrieve Player version
// Check if the string "WIN" is in version. If so,
// set isWin to true, otherwise set it to false.
isWin = (version.indexOf("WIN") != -1);
// If isWin is true...
if (isWin =  = true) {
  // ...perform Windows-specific actions here.
  trace("Please use IE5 or later on Windows.");
}
```

Experienced programmers will be quick to point out that we could have reduced *if (isWin = = true)* to *if (isWin)*, because isWin holds a Boolean value, which is what the *if* statement expects. While that's quite right, it doesn't make for a good example of the equality operator, now does it? Furthermore, new programmers usually find the more verbose form clearer. Both approaches are perfectly acceptable. We'll cover this topic in more detail in Chapter 7.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Chapter 5. Operators

5.5 The Strict Equality and Inequality Operators

We just saw that the standard equality operator (`= =`) can test whether two expressions have the same value. The *strict equality* operator, written as three consecutive equals signs (`= = =`), tests whether two expressions are of the same datatype before testing whether they have the same value.

For example, the following standard equality test yields `true` because the interpreter converts the string "13" to the number 13 before comparing the values:

```
trace("13" =  = 13);  // Displays: true
```

In contrast, the following strict equality test yields `false` because the operands are not of the same datatype:

```
trace("13" =  =  = 13);  // Displays: false
```

A strict equality test takes the general form:

```
operand1 =  =  = operand2
```

where *`operand1`* and *`operand2`* can be any valid expression. The strict equality operator determines the equality of its two operands as follows:

- If *`operand1`* and *`operand2`* are of different datatypes, they are not equal.
- If *`operand1`* and *`operand2`* are both `undefined`, they are equal.
- If *`operand1`* and *`operand2`* are both `null`, they are equal.
- If *`operand1`* and/or *`operand2`* is the special numeric value `NaN`, they are not equal.
- If *`operand1`* and *`operand2`* are both numbers and have the same numeric value, they are equal.
- If *`operand1`* and *`operand2`* are both strings and have the same characters in the same order, they are equal.
- If *`operand1`* and *`operand2`* are both `true` or both `false`, they are equal.
- If *`operand1`* and *`operand2`* store a reference to the same object, they are equal.
- Under any other condition, *`operand1`* and *`operand2`* are not equal.

The primary purpose of strict equality is to prevent the interpreter from performing datatype conversions when evaluating the equality of its two operands. For example, the following condition checks both whether `numLives` is a number and whether it is equal to 0:

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

```
if (numLives =  =  = 0) {
  trace("Game over");
}
```

which is more succinct than the alternative:

```
if ((typeof numLives =  = "number") && (numLives =  = 0)) {
  trace("Game over");
}
```

A strict inequality test takes the form:

```
operand1 !=  = operand2
```

The strict inequality operator (`!= =`) returns the Boolean opposite of the strict equality operator.

The strict equality operator, added in Flash MX, is created using three equals signs in a row (`= = =`). It should not be confused with either the standard equality operator (`= =`) or the assignment operator (`=`). Similarly, strict inequality, also added in Flash MX, is written as an exclamation point followed by two equals signs (`!= =`), while the standard inequality operator is written as an exclamation point followed by one equals sign (`!=`).

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Chapter 5. Operators

5.6 The Comparison Operators

The *comparison* operators (also called *relational* operators) are used to determine which of two values appears first in a given order. Like the equality and inequality operators, the comparison operators return one of the Boolean values, `true` or `false`, indicating whether the relationship described in the comparison is accurate (`true`) or inaccurate (`false`).

Comparison operators are intended for comparing strings and numbers; when used with other types, automatic datatype conversion occurs, as discussed later in this chapter. When the two operands of a comparison operator are numbers, the comparison is performed mathematically: *5 < 10* is `true`, *-3 < -6* is `false`, and so on. When the two operands of a comparison operator are strings, the comparison is performed according to character code points, as described in Chapter 4 under Section 4.6.2.2.

The interpreter attempts to convert any nonstring or nonnumeric data value used in a comparison operation to the *string* or *number* type. We'll consider the effect of datatype conversions on comparison operations after we discuss the comparison operators themselves.

5.6.1 The Less-Than Operator

The *less-than* operator takes the general form:

```
operand1 < operand2
```

If the operands are numeric, the less-than operator returns the Boolean `true` if *`operand1`* is mathematically smaller than *`operand2`*:

```
5 < 6          // true
5 < 5          // false; they are equal, but 5 is not less than 5
-3 < -6        // false; -3 is larger than -6
-6 < -3        // true;  -6 is smaller than -3
```

If the operands are strings, the less-than operator returns `true` if *`operand1`* comes before *`operand2`* in the Unicode character set; otherwise, it returns `false`:

```
"a" < "z"          // true; lowercase "a" comes before lowercase "z"
"A" < "a"          // true; uppercase letters come before lowercase
"Z" < "a"          // true; uppercase letters come before lowercase
"hello" < "hi"     // true; "e" comes before "i"
```

For further string comparison details, see Section 4.6.2.2 in Chapter 4.

5.6.2 The Greater-Than Operator

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

The *greater-than* operator takes the general form:

```
operand1 > operand2
```

If the operands are numeric, the greater-than operator returns the Boolean `true` if *`operand1`* is mathematically larger than *`operand2`*:

```
5 > 6          // false
5 > 5          // false; they are equal, but 5 is not greater than 5
-3 > -6        // true; -3 is greater than -6.
-6 > -3        // false; -6 is not greater than -3.
```

If the operands are strings, the greater-than operator returns `true` if *`operand1`* comes after *`operand2`* in the Unicode character set; otherwise, it returns `false`:

```
"a" > "z"          // false; lowercase "a" comes before lowercase "z"
"A" > "a"          // false; uppercase letters don't come after lowercase
"Z" > "a"          // false; uppercase letters don't come after lowercase
"hello" > "hi"     // false; "e" comes before "i"
```

For further string comparison details, see Section 4.6.2.2 in Chapter 4.

5.6.3 The Less-Than-or-Equal-to Operator

The *less-than-or-equal-to* operator takes the general form:

```
operand1 <= operand2
```

If the operands are numeric, the less-than-or-equal-to operator returns the Boolean `true` if *`operand1`* is mathematically smaller than or equal to *`operand2`*:

```
5 <= 6         // true
5 <= 5         // true; note the difference from 5 < 5
-3 <= -6       // false
-6 <= -3       // true
```

If the operands are strings, this operator returns `true` if *`operand1`* comes before *`operand2`* in the Unicode character set or if the operands are identical (according to the rules described in Section 4.6.2); otherwise, `<=` returns `false`:

```
"a" <= "z"          // true; lowercase "a" comes before lowercase "z"
"A" <= "a"          // true; although not equal, "A" comes before "a"
"Z" <= "a"          // true; uppercase letters come before lowercase
"hello" <= "hi"     // true; "e" comes before "i"
```

Note that the `<=` operator is written with the equals sign after the less-than sign. The following is not a valid operator: `=<`.

5.6.4 The Greater-Than-or-Equal-to Operator

The *greater-than-or-equal-to* operator takes the general form:

```
operand1 >= operand2
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

If the operands are numeric, the greater-than-or-equal-to operator returns the Boolean `true` if *`operand1`* is mathematically larger than or equal to *`operand2`*:

```
5 >= 6        // false
5 >= 5        // true; note the difference from 5 > 5
-3 >= -6      // true
-6 >= -3      // false
```

If the operands are strings, this operator returns `true` if *`operand1`* comes after *`operand2`* in the Unicode character set or if the operands are identical (according to the rules described in Chapter 4, Section 4.6.2); otherwise, `>=` returns `false`:

```
"a" >= "z"         // false; lowercase "a" comes before lowercase "z"
"A" >= "a"         // false; "A" comes before "a" and they are not equal
"Z" >= "a"         // false; uppercase letters come before lowercase
"hello" >= "hi"    // false: "e" comes before "i"
```

Note that the `>=` operator is written with the equals sign after the greater-than sign. The following is not a valid operator, although it does make a nice emoticon: `=>`.

5.6.5 Comparison Operations and Datatype Conversion

Most of the time, when we're using comparison operators, we're comparing numbers. Hence, type conversions instigated by the comparison operators favor numbers (just like the equality and inequality operators, discussed earlier). Note that this differs from the + operator, which favors conversion to the string datatype. When the two operands of any comparison operator belong to different datatypes, or when neither operand is a string or a number, the interpreter attempts a type conversion according to the following steps:

1. If both operands are numbers, compare the operands mathematically and return the result. If either number is (or if both numbers are) `NaN`, the result of the comparison is `false`, except in the case of the != operator.
2. If both operands are strings, compare the operands by Unicode code point and return the result. For characters in the Latin alphabet, this is a case-sensitive, alphabetic comparison.
3. If one operand is a number and the other is a string, convert the string to a number and go back to Step 1.
4. If either operand is a Boolean, `null`, or `undefined`, convert the operand to a number and go back to Step 1.
5. If either operand is an object, invoke its *valueOf ( )* method to convert the object to a primitive value and go back to Step 1. If the *valueOf ( )* method fails or does not return a primitive value, return `false`.
6. For any other condition, return `false`.

Note that type conversions performed during a comparison do not alter the original item's stored value or datatype. The results of the temporary conversions are discarded once the expression has been evaluated.

Here is a simple conversion example comparing two Booleans:

```
false < true       // true: 0 is less than 1
```

Comparison operators always convert composite datatypes to strings or numbers for comparison. In the following example, because both `someObj` and `someOtherObj` are members of the *Object* class, their string value, "[object Object]", is the same:

```
someObj = new Object();
someOtherObj = new Object();
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

```
someObj <= someOtherObj;    // true!
```

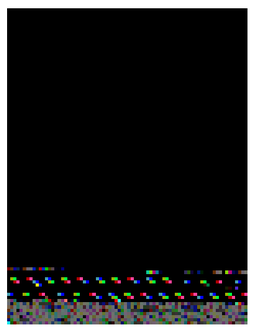
In contrast, the expression `someObj = = someOtherObj` yields `false` because objects are equality-tested by reference, and they are therefore not converted to primitives during an equality test.

In the next example, even though "A" has the code point 65, converting "A" to a number yields `NaN`, which means the whole expression yields `false`. Use the *charCodeAt( )* function to check a string's code point:

```
"A" <= 9999                  // false
"A".charCodeAt(0) < 9999     // true
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir



ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Chapter 5. Operators

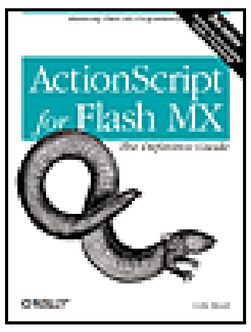
5.7 The Flash 4 String Operators

You might encounter the deprecated Flash 4 string operators—concatenation (`&`), equality (`eq`), inequality (`ne`), and comparison (`lt`, `gt`, `le`, `ge`)—when examining legacy code. The `==`, `!=`, `<`, `>`, `<=`, and `>=` operators should be used for all comparisons, including string comparisons, in Flash 5 and Flash MX. Likewise, use `+` for string concatenation in Flash 5 and Flash MX instead of the deprecated `add` operator or the `&` operator, which is the bitwise *AND* operator in Flash 5 and later.

For more details and a list of Flash 4 versus Flash 5 string-operation equivalencies, see Section C.3.6 in Appendix C.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir



ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Chapter 5. Operators

5.8 The Logical Operators

In Chapter 4, we saw how to make logical decisions using conditional expressions and Boolean values. The decisions were based on single factors: if a movie hasn't loaded, don't start playing; if a spaceship has double-shot power, increase the damag its shots inflict; and so on. Not all programming logic is so simple. We'll often want to consider multiple factors in our *branching* (i.e., decision-making) logic.

Suppose, for example, that we are making a personalized Flash site that requires users to log in. If a user logs in as a guest, th content he can see is limited. When a registered user logs into the site, we set a variable that indicates, for the duration of our movie, that the user is allowed to see privileged content:

```
var userStatus = "registered";
```

To decide whether to allow access to a restricted area, we use a Boolean check like this:

```
if (userStatus =  = "registered") {
  // It's okay, let 'em in...
}
```

Suppose we want to demonstrate the site to prospective investors without forcing them to register and without showing them material reserved for registered users. We can invent a new user category, "specialGuest", that allows investors to see more than a guest but not as much as a registered user. When we want to identify someone as a special guest, we set `userStatu` to "specialGuest".

How do we perform a Boolean check now that there are two categories of users? We could do things the brutish, painful way by duplicating portions of our code:

```
if (userStatus =  = "registered") {
  // Execute legal-user code...
}
if (userStatus =  = "specialGuest") {
  // Execute exact same legal-user code...
}
```

Obviously, that's going to turn into a real headache to maintain, because we have to duplicate our code changes in two places.

We'd prefer to perform a single Boolean check that says, "If the user is registered *or* if the user is a special guest, then grant access." We can produce compound logical phrases such as this with Boolean *logical operators* . Here, the logical *OR* operatc (`||`) lets us check the status of multiple Boolean operations in a single expression. I've included extra parentheses to make th order of operation more obvious:

```
if ( (userStatus =  = "registered") || (userStatus =  = "specialGuest") ) {
  // Execute legal-user code...
}
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Nice, huh? Sometimes programming is so elegant. Such distinctive vertical lines . . . such supple parentheses . . .

5.8.1 Logical Expressions and Data Values

Using logical operators to calculate a Boolean result is analogous to using arithmetic operators to calculate a numeric result. For example, the following four operands and three operators are used to calculate a single result:

```
(userStatus =  = "registered") || (userStatus =  = "specialGuest")
```

If the user is not registered, but is a special guest, this expression reduces to:

```
false || true
```

which in turn reduces to the Boolean result `true` .

Here is an analogous arithmetic operation that reduces to a single number:

```
(1 + 2) * (3 + 4)  // Four values (1, 2, 3, 4), which become...
3 * 7              // two values (3, 7), which become...
21                 // a single value (21)
```

Remember, a complex expression that reduces to a Boolean can be used anywhere a single Boolean value is required (such as the test expression of a conditional statement).

5.8.2 Logical OR

The *logical OR* operator is most commonly used to initiate some action when at least one of two conditions is met. For example, "If I am hungry *or* I am thirsty, I'll go to the kitchen." The symbol for logical *OR* is made using two vertical line characters (`||`). Typically, the `|` character is accessible using the Shift key and the Backslash (\) key in the upper right of mc Western keyboards, where it may be depicted as a dashed vertical line. Logical *OR* takes the general form:

```
operand1 || operand2
```

When both operands are Boolean expressions, logical *OR* returns `true` if either operand is `true` and returns `false` only i *both* operands are `false` . In summary:

```
true  || false   // true because first operand is true
false || true    // true because second operand is true
true  || true    // true (however, either operand being true is sufficient)
false || false   // false because both operands are false
```

This is a simplified case in which both operands are Booleans, but the operands can be any valid expression, including non-Boolean ones. In that case, the return value of a logical *OR* operation is not necessarily a Boolean (`true` or `false`). Technically, the first step in evaluating a logical *OR* is to convert *`operand1`* to a Boolean; if the result of such a conversio is `true` , logical *OR* returns *`operand1`* 's resolved value. Otherwise, logical *OR* returns *`operand2`* 's resolved value. Here's some code to demonstrate:

```
null || "hi there!"  // Operand1 does not convert to true, so the
                     // operation returns operand2's value: "hi there!"
true || false        // Operand1 is true, so the operation returns
                     // operand1's value: true.
"hey" || "dude"      // Operand1 is a nonempty string, so it converts to
                     // false and the operation returns
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

```
                        // operand2's value: "dude".
false || 5 + 5          // Operand1 does not convert to true, so the
                        // value of operand2 (namely 10) is returned.
```

In practice, we rarely make use of non-Boolean values returned by a logical *OR* expression. Instead, we normally use the resu in a conditional statement where it is used to make a Boolean decision. Consider Example 5-3.

Example 5-3. A logical OR operation as a conditional test expression

```
var x = 10;
var y = 15;
if (x || y) {
  trace("One of either x or y is not zero");
}
```

On line 3, we see a logical *OR* operation *(x || y)* being used where a Boolean is expected as the test expression of an *if* statement. Our first step in determining the value of *x || y* is to convert 10 (the value of the first operand, `x`) to a Boolean. As Table 3-3 shows, any nonzero finite number converts to the Boolean `true` . When the first operand converts to `true` , the logical *OR* returns the value of the first operand, in this case, 10. So, to the interpreter, our *if* statement looks like this:

```
if (10) {
  trace("One of either x or y is not zero");
}
```

But 10 is a number, not a Boolean. So what happens? The *if* statement converts the return value of the logical *OR* operation t a Boolean. In this case, 10 is converted to the Boolean value `true` (again, according to the rules in Table 3-3), and the interpreter sees our code as:

```
if (true) {
  trace("One of either x or y is not zero");
}
```

And there you have it. The test expression is `true` , so the *trace( )* statement between the curly braces is executed.

Note that if the first operand in a logical *OR* operation resolves to `true` , it is unnecessary, and therefore inefficient, to evaluate the second operand. Therefore, ActionScript evaluates the second operand only if the first operand resolves to `fals` . This fact is useful in cases in which you don't want to resolve the second operand unless the first operand resolves to `false` In the following example, we check if a number is out of range. If the number is too small, there is no need to perform the second test, in which we check whether it is too large.

```
if (xPos < 0 || xPos > 100) {
  trace ("xPos is not between 0 and 100 inclusive.");
}
```

Note that the variable `xPos` must be included in each comparison. The following code shows a common mistaken attempt to check `xPos` 's value twice:

```
// Oops! Forgot xPos in the comparison with 100
if (xPos < 0 || > 100) {
  trace ("xPos is not between 0 and 100 inclusive.");
}
```

5.8.3 Logical AND

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Like the logical *OR* operator, *logical AND* is used primarily to execute a block of code conditionally—in this case, only when both of two conditions are met. The logical *AND* operator takes the general form:

```
operand1 && operand2
```

Both `operand1` and `operand2` can be any valid expression. In the simplest case, in which both operands are Boolean expressions, logical *AND* returns `false` if either operand is `false` and returns `true` only if *both* operands are `true`. In summary:

```
true  && false   // false because second operand is false
false && true    // false because first operand is false
true  && true    // true because both operands are true
false && false   // false because both operands are false (either is sufficien
```

Let's see how the logical *AND* operator is used in two examples. In Example 5-4, we execute a *trace( )* statement only when two variables are both greater than 50.

Example 5-4. A logical AND operation as a conditional test expression

```
x = 100;
y = 51;
if (x>50 && y>50) {
  trace("Both x and y are greater than 50");
}
```

Because the expressions *x>50* and *y>50* are both `true`, the *trace( )* statement is executed.

In Example 5-5, we return to our web site example, in which access was restricted to registered users and special guests. But this time we'll let users in only if the date is not January 1. We combine expressions using the logical *OR* operator (to check the two allowed `userStatus` values) and the logical *AND* operator (to check the date). Note the parentheses that ensure the logical *OR* statement evaluates the `userStatus` before performing the logical *AND* with the date check. We determine the current date via the *Date* class, whose *getMonth( )* method returns 0 to indicate January.

Example 5-5. Compound logical expressions

```
userStatus = "registered";
now = new Date( );          // Create a new Date object
day = now.getDate( );       // Returns an integer between 1 and 31
month = now.getMonth( );  // Returns an integer between 0 and 11
// Note the indentation style and parentheses...if statements may span
// multiple lines for the sake of readability
if ( ((userStatus =  = "registered") || (userStatus =  = "specialGuest"))
    && (month + day) > 1) {
  // Let the user in...
}
```

The technical behavior of the logical *AND* operator is quite similar to that of the logical *OR* operator. First, `operand1` is converted to a Boolean. If the result of that conversion is `false`, the value of `operand1` is returned. If the result of that conversion is `true`, the value of `operand2` is returned.

If the first operand in a logical *AND* operation resolves to `false`, it is unnecessary, and therefore inefficient, to evaluate th

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

second operand. Therefore, ActionScript evaluates the second operand only if the first operand resolves to `true` . This fact i useful in cases in which you don't want to resolve the second operand unless the first operand resolves to `true` . In this example, we perform the division only if the divisor is nonzero:

```
if (numItems != 0 && total / numItems > 3) {
  trace("You've averaged more than 3 dollars per transaction");
}
```

5.8.4 Logical NOT

The *logical NOT* operator (`!`) returns the Boolean opposite of its single operand. It takes the general form:

```
!operand
```

where *`operand`* can be any legal expression. If *`operand`* is `true` , logical *NOT* returns `false` . If *`operand`* is `false` , logical *NOT* returns `true` . If *`operand`* is not a Boolean, its value is converted to a Boolean for the sake of the operation and its opposite is returned.

Like the does-not-equal operator (`!=`), the logical *NOT* operator is convenient for testing what something *isn't* rather than wh it *is* . Recall Example 5-5, in which we wanted to perform an action only if the date was not January 1. At first impression, this attempt might seem correct:

```
month != 0 && day != 1
```

We find, however, that *month != 0* is `true` anytime the month is not January (so far so good), but *day != 1* is actually `fals` on the first day of every month. So the expression *month != 0 && day != 1* would exclude our users on the first day of any month, not just January 1. This is hardly what we intended.

It's much easier to check if today *is* January 1:

```
month =  = 0 && day =  = 1
```

This expression yields `true` only on the first day of January. Once we've gotten that far, all we need is the *NOT* operator to determine when today is *not* January 1:

```
!(month=  =0 && day=  =1)
```

Obviously, this code's intent is much clearer than the *month + day > 1* test used in Example 5-5.

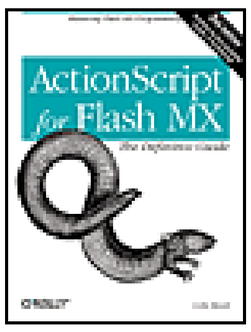
Another typical usage of the *NOT* operator is to *toggle* a variable from `true` to `false` and vice versa. For example, suppos you have a single button that is used to turn the sound on and off. You might use code like this:

```
soundState = !soundState  // Reverse the current sound state
if (soundState) {
  // If sound is turned on, make sure sounds are audible
} else {
  // If the sound is off, set the volume to 0 to mute it
}
```

Notice that `!` is also used in the inequality operator (`!=`). As a programming *token* (i.e., symbol), the `!` character usually means *not* , or *opposite* . It is unrelated to the ! symbol used to indicate "factorial" in common mathematical notation.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir



ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Chapter 5. Operators

5.9 The Grouping Operator

Aside from being used in function calls, parentheses—( )—can also be used to group a phrase of code to override the precede of various operators. (They also enhance legibility, even if they are not required to override the default precedence.) Parenthe are also required in some statements, most notably *if* statements. Parentheses take the general form:

```
(expression)
```

The *expression* within the parentheses is evaluated and returned. Here are some examples:

```
if (x =  = y) {                    // Syntactically required for if statement
  trace("x and y are equal"); // Required for the function call
}
(5 + 6) * 7                        // Force a nonstandard order of operation
(x >= 100) || (y <= 50)            // Parentheses not required but added for legibi
x >= 100 || y <= 50                // No parentheses...a little harder to read
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Chapter 5. Operators

5.10 The Comma Operator

Rarely used, the *comma* operator (`,`) allows us to evaluate two expressions where a single expression is expected. It takes the general form:

```
operand1, operand2
```

Both operands must be legal expressions. When a comma operation executes, `operand1` is evaluated, then `operand2` is evaluated, and then the resolved value of `operand2` is returned. In practice, the return value of the comma operator is usually ignored.

The comma operator is used primarily in *for* loops that initialize multiple variables, as shown in Example 5-6. Note that in the first line of the example, the expressions `i=0` and `j=10` are both operands of the comma operator. (These expressions cause `i` and `j` to be initialized to 0 and 10.) The expressions `i++` and `j--` are both operands of a second comma operator. (These expressions cause `i` to be incremented and `j` to be decremented each time the loop executes.) The comma operator is used to squeeze multiple expressions within the confines of the *for* loop's syntax.

Example 5-6. Using the comma operator in a for Loop

```
for (var i=0, j=10; i!=j; i++, j--) {
  trace ("i: " + i + " j: " + j);
}
// Which produces this output...
 i: 0 j: 10
 i: 1 j: 9
 i: 2 j: 8
 i: 3 j: 7
 i: 4 j: 6
```

The comma operator should not be confused with other uses of the comma, such as separating multiple arguments passed to a function or separating elements of an array.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Chapter 5. Operators

5.11 The void Operator

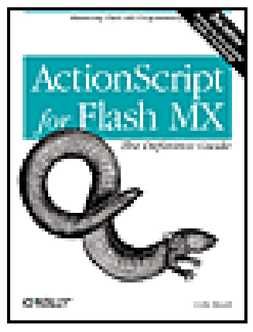
The *void* operator is used to evaluate an expression without returning a value. It has the syntax:

```
void expression
```

where *expression* is the expression to evaluate. In JavaScript, *void* is used to evaluate a JavaScript expression in a hypertext link without causing the result to appear in the browser's address bar. There are few, if any, such applications for *void* in ActionScript; it is supported for the sake of compatibility with ECMA-262.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir



ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Chapter 5. Operators

5.12 Other Operators

The remaining operators apply to topics considered in other chapters. We'll include them here for quick reference only and describe their usage fully in those chapters.

5.12.1 The typeof Operator

The *typeof* operator is used to determine the datatype of an expression or identifier. It takes one operand, as follows:

```
typeof operand
```

where `operand` can be any legal expression. The return value of the *typeof* operation is a string indicating the datatype of the evaluated `operand`. See Chapter 3 for more details.

5.12.2 The new Operator

The *new* operator creates a new composite datum—either an array or an object. The object can be a member of a built-in class or a user-defined class. The syntax for *new* is:

```
new constructor
```

where `constructor` must be a function that defines the properties of the newly created object. See Chapter 11, Chapter 12, and the *Constructor* headings in the *Language Reference*.

5.12.3 The delete Operator

We use the *delete* operator to remove an object, an object property, an array element, or variables from a script. The syntax for *delete* is:

```
delete identifier
```

If `identifier` is not a data container (i.e., a variable, property, or element), the *delete* operation fails and returns the value `false`; otherwise, it returns `true`, indicating success. The delete operator is commonly used to remove an event handler, as follows:

```
// Add an event handler:
_root.onEnterFrame = function ( ) {
  trace("A frame passed");
}

// Remove an event handler:
delete _root.onEnterFrame;
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

See Chapter 10, Chapter 11, and Chapter 12 for details on the *delete* operator.

5.12.4 Array-Element/Object-Property Operator

As we'll see in Chapter 11 and Chapter 12, we use the `[ ]` operator to retrieve and set the value of an array element or an object property. When accessing an array, it takes the form:

```
array[element]
```

where *`array`* is the name of an array or an array literal and *`element`* is an expression that resolves to a zero-relative, nonnegative integer representing the index of the array element to access.

When accessing an object, the `[ ]` operator takes the form:

```
object[property]
```

where *`object`* is an object name or object literal and *`property`* is any expression that resolves to a string representing the name of the object property to access.

When used on the left side of an assignment operator (`=`), the element or property is assigned the new value shown on the right side of the expression:

```
var colors = new Array();   // Create a new array
colors[0] = "orange";       // Set its first element
colors[1] = "green";        // Set its second element
var ball = new Object();    // Create a new object
var myProp = "xVelocity";   // Store a string in a variable
ball["radius"] = 150;       // Set the radius property
ball[myProp] = 10;          // Set the xVelocity property through myProp
```

When used anywhere else, the expression returns the value of the specified element or property:

```
diameter = ball["radius"] * 2;  // Sets diameter to 300
trace(colors[0]);               // Displays "orange"
```

5.12.5 The Dot Operator

The *dot* operator is our primary means of referring to object properties and nested movie clips. Functionally, the dot operator has the exact same purpose as the `[ ]` operator—it lets us set and retrieve object-property values. But the two operators have syntactic differences that make them unique. The general syntax of the dot operator is:

```
object.property
```

where *`object`* must be the name of an object or an object literal and *`property`* must be an identifier that represents a property of *`object`*. Note that *`property`* cannot be an arbitrary expression or a string literal; it must be the name of a property. Because array elements are numbered, not named, the dot operator cannot be used to access the elements of an array.

When used as the left-hand operand of an assignment operator, the dot operator is used to set a new value for a property:

```
var ball = new Object();
ball.radius = 150;
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

```
ball.xVelocity = 10;
```

When used anywhere else, a dot operation returns the value of the named object property:

```
diameter = ball.radius * 2;
newX = ball.xPosition + ball.xVelocity;
```

The dot operator is also used to invoke object methods (functions stored in properties), as follows:

```
intro_mc.gotoAndPlay("beginMovie");
```

See Chapter 12 and Chapter 13 for details on the dot operator.

5.12.6 The Conditional Operator

The *conditional* operator is a syntactic convenience that lets us succinctly represent a simple conditional statement. This operator takes three operands, in the form:

```
condition ? result_if_true : result_if_false;
```

When a conditional operation is executed, the first operand (`condition`) is evaluated. If `condition` is `true` or can be converted to `true`, the value of the second operand (`result_if_true`) is returned. Otherwise, the value of the third operand (`result_if_false`) is returned. See Chapter 7 for details on the conditional operator.

5.12.7 The instanceof Operator

The *instanceof* operator, added in Flash MX, checks whether an object is an instance of a given class (or one of the class's subclasses). It takes the form:

```
object instanceof classConstructor
```

where `object` is any object and `classConstructor` is a reference to any class's constructor function. The operation returns `true` if `object` is an instance of `classConstructor` or one of its subclasses; otherwise, it returns `false`. For example, the following code creates a *Date* object and then checks whether the object is an instance of the *Date* class:

```
// Create an instance of a built-in class.
var now = new Date( );
trace(now instanceof Date);  // Displays: true
```

The following code creates a *Shape* class with a *Circle* subclass and then checks whether a *Circle* object is an instance of *Shape*:

```
// Create Shape and Circle classes (constructor body code omitted)
function Shape ( ) {
  // ...
}
function Circle ( ) {
  // ...
}
// Establish inheritance: Circle is a subclass of Shape
Circle.prototype = new Shape( );
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

```
// Create a Circle object
var theCircle = new Circle( );
// Check the class of the Circle object using instanceof
trace(myCircle instanceof Circle);  // true
trace(myCircle instanceof Shape);   // true
```

The *instanceof* operator can be used to operate on an object conditionally, based on its class. For example, the following code uses *instanceof* to remove all text fields and movie clips from the current movie clip (`this`):

```
for (var p in this) {
  if (this[p] instanceof MovieClip) {
    this[p].removeMovieClip( );
  } else if (this[p] instanceof TextField) {
    this[p].removeTextField( );
  }
}
```

Note that the *instanceof* operator simply checks whether `object`'s prototype chain includes `classConstructor`, as discussed in Chapter 12 under Section 12.7.

5.12.8 The super "Operator"

Added in Flash MX, *super* is an operator-like tool that serves two distinct object-oriented programming purposes:

- It can invoke an object's superclass constructor.
- It can invoke a superclass's implementation of an overridden method.

In either case, *super* can be used only within a class constructor function or a method.

To invoke an object's superclass constructor function, use the following syntax:

```
super(arg1, arg2,...argn);
```

where `arg1,...argn` is a list of arguments passed to the superclass constructor function. Within the superclass constructor function, the `this` keyword points to the object whose class or method invoked the constructor. Used in this manner, the *super* operator is functionally equivalent to the following statement:

```
this.constructor.prototype.constructor.apply(this, arguments);
```

To invoke a superclass method, we use the following syntax:

```
super.methodName(arg1, arg2,...argn);
```

where `methodName` is some method of the superclass and `arg1,...argn` is a list of arguments passed to that method. Within the method, the `this` keyword points to the object whose class or method invoked the superclass method. Used in this manner, the *super* operator is functionally equivalent to the following rather unwieldy statement:

```
this.constructor.prototype._ _proto_ _.constructor.apply(this, arguments);
```

For practical examples showing *super* in use, see Section 12.5.2 and Section 12.5.3.2 in Chapter 12.

Note that *super* is not technically an operator. Rather, it is an identifier made available only during function execution,

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

much like the `arguments` object. ActionScript's *super* is not an implementation of ECMAScript v4's *super* operator. Nevertheless, for the sake of discussion, both Macromedia's *ActionScript Dictionary* and this book categorize *super* with other operators.

5.12.9 The Function Call Operator

As we've seen with the *trace( )* function and the string-manipulation functions, we use the function call operator, `( )`, to invoke a function. A function call takes the general form:

```
function_name(argument_list)
```

The first operand, *`function_name`*, is the name of some function and must be an identifier, not an expression. If the function does not exist, the interpreter will *not* produce an error but merely fails silently. The *`argument_list`* is a series of zero or more arguments passed to the function, separated by commas. The return value of a function call operation is the return value supplied by the function itself.

The function call operator can call any built-in or user-defined function:

```
trace("an internal function");    // Built-in function, one argument
theClip_mc.play( );              // Method of a movie clip, no arguments
myRectangle.area(6, 9);           // User-defined method, two arguments
init( );                          // User-defined function, no arguments
```

5.12.10 The Bitwise Operators

ActionScript supports a group of so-called *bitwise operators* that manipulate the binary digits (*bits*) of an integer. If you're planning to develop large-scale systems in which every iota of memory, calculation speed, and transfer-rate optimization makes a meaningful difference in performance, read about the bitwise operators online at:

http://www.moock.org/asdg/technotes/bitwise

Otherwise, use the Boolean logical operators (covered earlier in this chapter) instead. They perform the same tasks as the bitwise operators, albeit in a less optimized way. For quick reference, the bitwise operators (`<<`, `>>`, `>>>`, `~`, `&`, `^`, and `|`) are included in Table 5-1.

See the *Color.getRGB( )* method in the ActionScript Language Reference for an example that uses the bitwise operators to extract the red, green, and blue components from an RGB color triplet.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Chapter 5. Operators

5.13 Onward!

Like all languages, ActionScript has a grammar and different parts of speech. Over the last few chapters, you've learned the equivalent of sentence structure, nouns, adjectives, and conjunctions. Now, let's move on to statements, which, like verbs, can do things. Statements will complete our ability to form sentence-like instructions.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Part I: ActionScript Fundamentals

Chapter 6. Statements

We saw in earlier chapters how ActionScript stores and manipulates data. In this chapter we'll see how to do things with that data. We'll give Flash instructions in the form of *statements*, or phrases of code that instruct the interpreter to perform a specified task.

To make something happen in Flash—whether stopping a sound, playing a movie, running a function, or repeatedly executing some code—we use a statement. In fact, an ActionScript program can be defined as nothing more than a list of statements that tell the interpreter what we want Flash to do. Here, for example, is an entire ActionScript program, which consists of four statements and tells the interpreter to load a web page into the browser:

```
var protocol = "http";                                  // Statement 1
var domain = "www.moock.org";                           // Statement 2
var path = "webdesign/flash/";                          // Statement 3
getURL(protocol + "://" + domain + "/" + path);         // Statement 4
```

Scripting a movie with ActionScript is simply a matter of attaching statements to frames, movie clips, and buttons. Unless instructed otherwise, Flash performs each statement in the order in which it appears in the script (later we'll explore how and when Flash decides which script to execute). This chapter explores the syntactic makeup of statements and lists the general statement categories. We'll touch on all the ActionScript statements in this chapter, but some of the more important ones will be examined in detail in later chapters.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Chapter 6. Statements

6.1 Types of Statements

Conceptually speaking, there are five core types of statements:

Statements that control the execution flow of a program
- loops
- conditionals
- ifFrameLoaded

Statements that declare variables
- var
- set

Statements that declare, call, and return values from functions
- function
- function call
- return

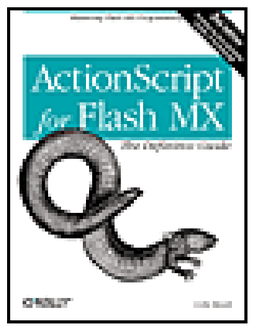
Statements that deal with objects
- with
- for . . . in

Statements that represent a data value
- any expression (especially expressions with side effects)

These informal categories help us understand what we can tell the interpreter to do using statements. At first glance, the list may seem fairly limited. However, we'll see that there are many variations of conditionals and loops and that there are thousands of things we can do with function calls. First, let's see how statements are formed.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir



ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Chapter 6. Statements

6.2 Statement Syntax

Statements are typically made up of *keywords* and *expressions*. We've already seen the *var* statement, which declares, and optionally, initializes a variable. Here's an example of the *var* statement's general syntax:

```
var numFrames;
```

The *keyword* (*var* in this case) begins the statement. Each keyword is followed by the syntax (parentheses, parameters, operands, etc.) required to complete the statement, which, in our example, is simply the name of a variable, `numFrames`. Finally, a semicolon marks the end of the statement.

In ActionScript, every statement should end with a semicolon, which is not to say that there is one semicolon per line of the script. Sometimes there are multiple statements on a line, each deserving their own semicolon. Sometimes a statement continues onto more than one line, and the semicolon doesn't appear until the end of the statement. You don't need semicolons after the curly braces that sometimes enclose statements, except in the special case of a function literal. It's good form to use semicolons, but it's not mandatory. We'll study semicolon usage in Chapter 15.

Some statements can take multiple forms. For example, we can use *var* with or without an optional initial assignment (in this case, 10 is a simple expression whose value is assigned to `x`):

```
var x;        // Simple declaration
var x = 10;   // Declaration with assignment
```

We'll discuss the specific syntax of each statement throughout the rest of this chapter.

6.2.1 Statement Blocks

Some statements actually include other statements, or *substatements*, as part of their syntax. For example, the *if* statement has this syntax:

```
if (expression) substatement;
```

The *`substatement`*, which is executed only if *`expression`* evaluates to `true`, can be a single statement, such as a variable declaration statement:

```
if (x =  = 5) var numFrames = 2;
```

or it can be a series of statements grouped together as a *statement block*:

```
if (x =  = 5) {
  var numFrames;
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

```
  numFrames = 10;
  play();
}
```

As you can see, a statement block is any number of statements on one or more lines, surrounded by curly braces. Here we show three statements, separated by semicolons, all on one line:

```
{ statement1; statement2; statement3... }
```

By using a statement block as the substatement of our *if* statement, we can specify multiple statements to be executed conditionally (only when the *if* statement is `true`). This can be very handy.

We can use a statement block anywhere ActionScript expects a single statement. In fact, statement blocks are sometimes required. For example, the *function* statement must always include a statement block, even if the function being declared has only one statement in its body:

```
function aheadTwoFrames ( ) {
  gotoAndStop(_currentframe + 2);
}
```

For the sake of readability, statement blocks are typically formatted as shown here:

```
parent_syntax {
  substatement1;
  substatement2;
  substatement3;
}
```

where *`parent_syntax`* represents the statement for which we are defining a statement block.

Note that the opening curly brace appears at the end of the first line after *`parent_syntax`*. The substatements of the statement block appear indented two spaces, each on its own line. The closing curly brace, also on its own line, is flush with the main statement. Substatements within statement blocks should end with a semicolon, but there are no semicolons on the lines with curly braces.

The indenting format is simply a convention, not a requirement of legal syntax. This book adheres to the style used by the Flash ActionScript editor (the Actions panel). The following format is also valid and common:

```
parent_syntax
{
  substatement1;
  substatement2;
  substatement3;
}
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Chapter 6. Statements

6.3 The ActionScript Statements

Now that you know how a typical statement is formed, skim Table 6-1 to acquaint yourself with some of the things that ActionScript statements do.

Table 6-1. The ActionScript statements

Statement	Syntax	Use
break	`break;`	Aborts a loop or *switch* statement
case	`case expression:` `  *statements*`	Identifies a statement to be executed conditionally in a *switch* statement
continue	`continue;`	Skips the remaining statements in the current loop and begins the next iteration at the top of the loop
default	`default:` `  *statements*`	Identifies the statement(s) to execute in a *switch* statement when the test expression does not match any *case* clauses

do-while	`do {`
	`  *statements*`
	`} while (*expression*);`
		A variation of a *while* loop that ensures at least one iteration of the loop is performed

empty statement	`;`	Holds a place where a statement is expected; used with *evaluate* in Normal Mode

for	`for (*init*; *test*; *increment*) {`
	`  *statements*`
	`}`
		Executes a statement block repetitively (a *for* loop)

for-in	`for (*property* in *object*) {`
	`  *statements*`
	`}`
		Enumerates the properties of an object

function	`function *name* (*parameters*) {`
	`  *statements*`
	`}`
		Declares a function

if-else if-else	`if (*expression*) {`
	`  *statements*`
	`} else if (*expression*) {`
	`  *statements*`
	`} else {`
	`  *statements*`
	`}`
		Executes one or more statements, based on a condition or a series of conditions

ifFrameLoaded	`ifFrameLoaded (*frame*) {`
	`  *statements*`
	`}`
		Executes specified code if a certain frame has loaded; deprecated as of Flash 5

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

return	`return;` `return` *`expression`*`;`	Exits and optionally returns a value from a function
set	`set(`*`variable, value`*`);`	Assigns a value to a dynamically named variable

switch	`switch (`*`expression`*`) {`
	`  `*`statements`*
	`}`
		Executes specified code, based on a condition or a series of conditions (alternative to if-else if-else)

var	`var` *`variableName`*`;` `var` *`variableName = expression`*`;`	Declares and optionally initializes a variable

while	`while (`*`expression`*`) {`
	`  `*`statements`*
	`}`
		Executes a statement block repetitively (a *while* loop)

with	`with (`*`objectName`*`) {`
	`  `*`statements`*
	`}`
		Executes a statement block in the scope of a given object

6.3.1 Loops and Conditionals

We've already had several informal encounters with loops and conditionals. Together, these two statement types account for the majority of complex flow in our programs. Loops allow us to execute statement blocks repeatedly, and conditionals allow us to execute statement blocks under only the specified circumstances. See Chapter 7 and Chapter 8 for complete details.

6.3.2 Expression Statements

Although any expression can be used independently as a valid statement, if the expression performs no action and we don't do anything with the result, the exercise is rather pointless:

```
"hi there";  // This doesn't do much
345 + 5;     // Neither does this
```

Some expressions, however, have side effects that change the state of a system through variable or property assignments or by physically altering the Flash environment. In this example, an expression changes the value of `x` :

```
x = 345 + 5;  // Much more useful
```

Function calls are the most powerful type of expression statement. Even if a function doesn't return a useful value, it may have a very useful side effect. For example, *gotoAndStop( )* returns the value `undefined` but has an important side effect—it moves the playhead to another frame:

```
gotoAndStop(5);  // The value of _currentframe is changed to 5.
```

We'll learn more about function calls in Chapter 9 under "Running Functions."

6.3.3 The var Statement

The *var* statement declares a new variable:

```
var x;
```

You can assign the new variable's initial value as part of a *var* statement:

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

```
var x = 10;
```

When used outside a function, the *var* statement creates a variable scoped to the timeline containing the statement. When used inside a function, the *var* statement creates a variable local to that function (i.e., one that dies when the function finishes). Refer back to Chapter 2 for more information on variables.

6.3.4 The set Statement (Set Variable)

For most variable assignments, we use an assignment expression in statement form, like this:

```
x = 30;
```

This kind of expression, however, requires us to know the name of our variable in advance. If we want to generate the name of a variable dynamically during an assignment, we can use the *set* statement, which has the following syntax:

```
set(variable, expression);
```

where `variable` is a string expression to be used as the variable name in the assignment and `expression` is the new value to assign to it. For example:

```
var x;
set("x", 10);                    // x is now 10
var firstName;
set("first" + "Name", "jane");  // firstName is now "jane"
```

In the following, trickier example, `y` is not set in the *set* statement. Instead, `y` 's value (`"x"`) is retrieved and that value is used as the name of the variable to set:

```
// Pay close attention to this one...
var y = "x";
var x;
set(y, 15);  // x is now 15, y is still "x".
```

In Flash 4, *set* was called *Set Variable* , and it was often used to dynamically assign variables that had programmatically generated sequential names. It allowed programmers to simulate arrays, which were not a native part of Flash 4 ActionScript. Since arrays were added in Flash 5, *set* is rarely used anymore. In most cases, a variable can be referenced dynamically with the object property access operator, `[ ]` . For example,

```
// Set the value of firstName in the form_mc movie clip.
form_mc["first" + "Name"] = "jane";

// Set the value of firstName in the current movie clip.
this["first" + "Name"] = "jane";
```

However, *set* is required when dynamically referring to a local variable in a function (because local variables are not properties of any exposed ActionScript object).

6.3.5 The function Statement

Just as the *var* statement is used to *declare* (i.e., create) variables, the *function* statement is used to declare functions. The *function* statement has the following syntax:

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

```
function funcName (param1, param2, param3,...paramn) {
  statements
}
```

The keyword `function` begins the statement; *`funcName`* is the name of the function being declared; *`param1`* through *`paramn`* define the parameters required by the function when it executes (parameters are separated by commas); *`statements`* is a list of one or more statements that will be executed when the function is called.

The *function* statement declares a function for later use but does not execute that function. To execute a function, we use the function's name in a *function call* statement, which we'll discuss next.

6.3.6 Function Call Statements

A function call statement executes a built-in or user-defined function simply by using the function's name and providing the inputs that the function needs to perform its job. The terms *call* , *run* , and *invoke* are often used interchangeably to mean that a function's name has been used to cause it to execute. Function call statements have the following general syntax:

```
funcName(arg1, arg2, ... argn);
```

where *`funcName`* is the name of the function we want to execute and *`arg1`* through *`argn`* is the list of *arguments* (i.e., input values) that the function expects when it runs.

The function call statement is an extremely powerful and fundamental device; it's our primary means of controlling Flash movies. When we want to manipulate a movie in some way, we often call a function. Here are a few examples:

```
play();                          // Plays the current movie
gotoAndStop(5);                  // Sends the playhead to frame 5
startDrag("crosshair", true);    // Makes the "crosshair" instance follow
                                 // the mouse pointer
```

Function calls are also used with objects to invoke methods:

```
circle.getArea();
today.getDate();
```

Because movie clip instances are objects, we frequently use method invocation in our scripts:

```
ball_mc.play()
intro_mc.gotoAndStop("end");
```

We'll cover the use of functions in Chapter 9, and in Chapter 12 we'll see how functions can become object methods.

6.3.7 The return Statement

When we call a function, we can optionally pass it one or more values (*parameters* or *arguments*) to manipulate during execution. Likewise, a function can send us back a *return value* (a value that results from the execution of a function and is sent back to the caller). Within the body of a function, we use the *return* statement to conclude the function's execution and optionally return a value. A *return* statement takes one of the following forms:

```
return;
return expression;
```

The optional *`expression`* , if included, becomes the return value of the function. A *return* statement without a return

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

value exits the function and returns the value `undefined` . All *return* statements exit the current function. Note that *return* statements are not required in functions; a function without a *return* statement simply ends after the last statement in the function body and implicitly returns `undefined` . See Chapter 9 for more details on creating, calling, and terminating functions.

6.3.8 The with Statement

The *with* statement provides a shorthand way to refer to properties of an object without having to retype the object's name repeatedly. A *with* statement takes the general form:

```
with (object) {
  statements
}
```

When a property is referenced within a *with* statement block, `object` is checked for the specified property. In other words, *with* temporarily adds `object` to the end of the scope chain for the currently executing code. If the property exists in `object` , then `object` 's property is used to resolve the property reference. If the property does not exist in `object` , the remainder of the current scope chain is consulted for the property in question, as described in Chapter 2.

The following example shows the difference between executing a statement inside a *with* statement and outside a *with* statement:

```
PI = 10;                    // Set a timeline variable, PI
with (Math) {               // Execute statements in the context of Math
  trace("pi is: " + PI);    // Displays: 3.1459... (PI is a property of Math)
}
trace("PI is: " + PI);      // Displays: 10 (Math is no longer accessed)
```

In addition to providing convenient access to object properties, *with* can be used to invoke object methods:

```
x = 10;
y = 11;
with (Math) {
  larger = max(x, y);
}
trace(larger);  // Displays: 11
```

It is not possible to define a new property on an object that is the target of a *with* statement. For example, in the preceding code, the variable `larger` does not exist on the *Math* object, so the property assignment creates a new variable on the timeline that contains the *with* statement (not on *Math* !). The following code shows a misguided attempt to set a variable in `theClip_mc` :

```
with (theClip_mc) {
  var x = 10;  // x is set on the current timeline, not theClip_mc
}
```

We can, however, legitimately use *with* to affect movie clip instances in other ways. It can provide a handy way to work with deeply nested instance structures. For example, we can change this:

```
this.player_mc.eyes_mc.blinkRate = 30;
this.player_mc.eyes_mc.gotoAndStop("eyesOpen");
```

to this:

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

```
with (this.player_mc.eyes_mc) {
  blinkRate = 30;            // Resets an existing variable in eyes_mc instance
  gotoAndStop("eyesOpen");  // Function applied to eyes_mc instance
}
```

But *with* is not our only means of achieving this convenience. We can also simply assign our instance to a variable and use that variable in our references:

```
var eyes = this.player_mc.eyes_mc;
eyes.blinkRate = 30;
eyes.gotoAndStop("eyesOpen");
```

Many developers find the variable approach easier to read and work with than the *with* statement, but both are valid. We'll learn more about treating movie clips as objects in Chapter 13.

Note that inside a *with* statement, the value of the `this` keyword does not change; it retains the value it had outside the *with* statement. For example:

```
// Place the following code on _level0
// On _level0, create a ball object with a radius property
_level0.ball = { radius: 20 };
// Check the value of 'this' outside of the with statement
trace(this);  // Displays: _level0
// Start our with statement
with (_level0.ball) {
  // Check the value of 'this' inside the with statement
  trace(this);   // Still displays: _level0 (not ball!)
  // Check the radius of the ball object
  trace(radius); // Displays: 20
}
```

6.3.9 The ifFrameLoaded Statement

The deprecated *ifFrameLoaded* statement executes its substatements only if the specified frame has been loaded. Here's the syntax:

```
ifFrameLoaded (frame_expression) {
  statements
}
```

where *`frame_expression`* must be either the number of a frame or a string indicating a frame label. If the frame indicated by *`frame_expression`* has downloaded to the Player, *`statements`* are executed; if not, the statement block is skipped.

Because *ifFrameLoaded* lacks an *else* clause, it was deprecated in Flash 4. In Flash 4 and later, you should use the `_totalframes` and `_framesloaded` properties with *if-else* statements to create a more versatile preloader. For example:

```
if (_totalframes > 0 && _framesloaded =  = _totalframes) {
 gotoAndPlay("beginMovie");
} else {
 gotoAndPlay(_currentframe - 1);
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

```
}
```

6.3.10 The Empty Statement

For the sake of completeness, we should mention that it is legal to issue a statement with no content using a lone semicolon:

```
;
```

The *empty statement* has very little practical application, except that it can be used as a placeholder anywhere a statement is normally expected. It is not needed if you simply want to add blank lines to your code. ActionScript ignores blank vertical lines, which can be used to improve code readability.

In Flash's Normal Mode, the lone semicolon appears when the *evaluate* Action is added to a block of code. An arbitrary statement can then be added by typing into the Expression field of the Parameters panel.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

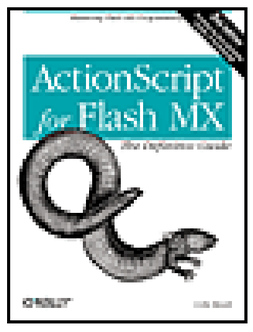
Chapter 6. Statements

6.4 Statements Versus Actions

If you look through the Flash ActionScript editing environment, you won't find any reference to the word "statement." Even Macromedia's documentation uses the terms "Action" and "statement" interchangeably, claiming, "Actions are statements that instruct a movie to do something while it is playing." Using the term "Action" as a synonym for "statement" blurs the distinction between several different ActionScript tools. To see how, open the Actions panel (F9) and look in the Actions folder. Under the list of Actions in that folder (and its subfolders), you'll find the statements we saw earlier in Table 6-1. Interspersed with the statements, you'll also notice quite a few functions: *gotoAndPlay( )*, *getURL( )*, *startDrag( )*, and so on. Although the functions listed as Actions can be used in statements, technically they're not unique statement types—they're just built-in functions. Statements, some built-in functions, and event handlers are all called "Actions" by Macromedia. Throughout this book, we do not use the generic term "Action". Instead, each Action is described with the term that matches its formal role in the language: either *statement*, *function*, or *event handler*. Understanding ActionScript from this traditional perspective makes it easier to migrate your skills to other languages and to understand general programming discussions that do not relate specifically to ActionScript.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir



ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Chapter 6. Statements

6.5 Onward!

Though you've learned a lot about statements, two important statement types—conditionals and loops—are yet to come. We'll deal with these kinds of statements in the next two chapters.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Part I: ActionScript Fundamentals

Chapter 7. Conditionals

Most of the code examples we've seen so far have been very *linear*—each statement is executed in turn, starting with the first and ending with the last. Linear code always does the same thing. A *conditional*, in contrast, is a type of statement that performs an action only when a specified condition is met. In linear code, the interpreter might execute statement A, then statement B, then statement C. With conditional statements, we can tell the interpreter to execute statement A, then execute *either* statement B *or* statement C, depending on condition X.

We can use conditionals to create and control situations that have more than one potential outcome. For example, suppose we want to create a password-protected site. When users attempt to log in, either the password is correct and the user is allowed to enter the site, or the password is wrong and the user sees an error message. The two outcomes require two very different blocks of code in our movie's script. One block needs to send the Flash playhead to a frame containing the site's welcome screen, and the other block needs to send the playhead to a frame with an error message. But only one of the blocks should be executed when the user attempts to log in. A conditional statement allows us to execute the appropriate block and skip the inappropriate one.

How does the interpreter know which code block to execute? When we define a conditional statement, we specify the condition that must be met in order for the *first* block of code to be executed. If the condition is not met, an alternate block of code may be executed (and that alternate block may, in turn, have its own condition). Essentially, we set up flowchart-like logic in our program that, in pseudocode, reads like this:

```
if (the first condition is met) {
  // Execute this code if the first condition is met
} else if (the second condition is met) {
  // Execute this code if the second condition is met
} ...otherwise {
  // Execute this code if none of the earlier conditions were met
}
```

Of course, we must describe each condition in terms the interpreter understands. This isn't a problem—it's just a question of syntax, which we'll cover next. Conceptually, all conditionals either allow or suppress the execution of a code block, based on the specified condition. Now let's see how conditionals work in practice.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Chapter 7. Conditionals

7.1 The if Statement

The *if* statement is your everyday, all-purpose conditional. We can use an *if* statement to create a two-pronged branch in our code, like a fork in the road. The *if* statement contains one or more substatements that are executed only when the specified condition is met. The *if* statement has the following syntax:

```
if (condition) {
  substatements
}
```

An *if* statement starts, not surprisingly, with the keyword `if`. The `condition` that must be satisfied for `substatements` to be executed is enclosed in parentheses. The `substatements` are one or more ActionScript statements. Each substatement should be on its own line and terminated with a semicolon. The entire *if* statement ends with a closing curly brace (`}`), without a trailing semicolon.

The `condition` of our *if* statement can be any valid expression. When an *if* statement is executed, the interpreter checks the value of that expression (called the *test expression*). If it is `true`, the `substatements` are executed. Otherwise, the `substatements` are not executed. Here we use simple Boolean values as the test expression:

```
if (true) {
  trace("The condition was met!");  // This statement will be executed
}
if (false) {
  trace("The condition was met!");  // This statement is never executed
}
```

Of course, there's no practical reason to use Boolean literals as test expressions, because their values never change. Instead, we'll use complex expressions that *return* Boolean values. For example, expressions that involve a comparison operation return a Boolean value, which makes them perfect for conditional test expressions:

```
var pointerX = _root._xmouse;  // Horizontal location of the mouse
// If pointerX > 300 yields true...
if (pointerX > 300) {
  // ...this statement is executed
  trace("The mouse is past the 300 pixel mark");
}
```

Now for the cool part: the test expression of a conditional doesn't necessarily have to evaluate to a Boolean—any expression will do. We can use a string or a number as the test expression of a conditional:

```
if ("hi") {
  trace("The condition was met!");
}
if (4) {
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

```
  trace("The condition was met!");
}
```

How does this work if the expressions "hi" and 4 are not Booleans? The answer lies in the marvels of datatype conversion, as shown in Table 3-3. When the test expression of a conditional statement is not a Boolean value, the interpreter converts the expression to a Boolean. For example, the interpreter converts "hi" to `false` because all nonnumeric strings convert to `false` when used in a Boolean context. So the condition is not met and the first *trace( )* statement is not executed. Similarly, the interpreter converts the number 4 to `true` (any nonzero number converts to `true`), so the second *trace( )* statement is executed.

All our earlier work with datatype conversion has paid off! Here are some basic applied examples. Try to guess whether each substatement will be executed:

```
x = 3;
if (x) {
  trace("x is not zero");
}
```

This example uses the *OR* operator, described in Chapter 5:

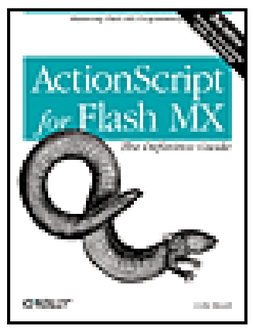
```
lastName = "";
firstName = "";
if (firstName != "" || lastName != "") {
  trace("Welcome " + firstName + " " + lastName);
}
```

Finally, we test whether a movie clip object exists (movie clips are converted to `true` when used in a Boolean context):

```
if (theClip_mc) {
  theClip_mc._x = 0;  // If theClip_mc exists, put it on
                      // the left edge of the Stage
}
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir



ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Chapter 7. Conditionals

7.2 The else Statement

With a lone *if* statement, we can cause a single code block to be optionally executed. By adding an *else* clause, we can choose which of two code blocks should be executed. Syntactically, an *else* statement is an extension of an *if* statement:

```
if (condition) {
  substatements1
} else {
  substatements2
}
```

where `condition` can be any valid expression. `substatements1` will be executed if `condition` is `true`; `substatements2` will be executed if `condition` is `false`. In other words, an *else* statement is perfect for depicting a mutually exclusive decision; one code block will be executed and the other will not.

Here's some code to demonstrate:

```
var lastName = "Grossman";
var gender = "male";
if (gender =  = "male") {
  trace("Good morning, Mr. " + lastName + ".");
} else {
  trace("Good morning, Ms. " + lastName + ".");
}
```

The *else* clause often acts as the backup plan of an *if* statement. Recall our password-protected web site example. If the password is correct, we let the user enter the site; otherwise, we display an error message. Here's some code we can use to perform the password check (assume that `userName` and `password` are the user's entries and that `validUser` and `correctPassword` are the correct login values):

```
if (userName =  = validUser && password =  = correctPassword) {
  this.gotoAndPlay("intro");
} else {
  this.gotoAndStop("loginError");
}
```

7.2.1 The Conditional Operator

Simple two-part conditional statements can be expressed conveniently with the conditional operator (`?:`). The conditional operator has three operands, which you'll recognize as analogous to the components of an *if-else* statement series:

```
condition ? expression1 : expression2;
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

In a conditional operation, if *`condition`* is `true`, *`expression1`* is evaluated and returned. Otherwise, *`expression2`* is evaluated and returned. The conditional operator is just a quicker way to write the following conditional statement:

```
if (condition) {
  expression1
} else {
  expression2
}
```

Like a conditional statement, the conditional operator can be used to control the flow of a program:

```
// If command is "go", play the movie; otherwise, stop
command =  = "go" ? play( ) : stop( );
```

The conditional operator also provides a terse way to assign values to variables:

```
var guess = "c";
var answer = "d";
var response = (guess =  = answer) ? "Right" : "Wrong";
```

which is equivalent to:

```
var guess = "c";
var answer = "d";
if (guess =  = answer) {
  response = "Right";
} else {
  response = "Wrong";
}
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Chapter 7. Conditionals

7.3 The else if Statement

Using *if* and *else*, we can optionally execute one of two code blocks. By using *if* and *else if*, we can optionally execute one (or even none) of an *unlimited* number of code blocks. Like *else*, *else if* is a syntactic extension of an *if* statement:

```
if (condition1) {
  substatements1
} else if (condition2) {
  substatements2
} else if (condition3) {
  substatements3
} else {
  substatements4  // Catchall if other conditions were not met
}
```

where `condition1`, `condition2`, and `condition3` must be valid expressions. `substatements1` will be executed if `condition1` is `true`. If `condition1` is `false`, `substatements2` will be executed if `condition2` is `true`. Otherwise, `condition3` is evaluated, and the process continues for as many *else if* statements are provided. If none of the test expressions are `true`, the statements in the final catchall *else* clause will be executed. For example, we could write a login-checking routine to provide insightful error messages, like this:

```
if (userName != validUser) {
  message = "User not found. Please try again.";
  this.gotoAndStop("loginError");
} else if (password != correctPassword) {
  message = "Password incorrect. Please try again.";
  this.gotoAndStop("loginError");
} else {
  this.gotoAndPlay("intro");
}
```

Note that an *else if* statement is merely a combination of an *else* statement with a nested *if* statement. Although the following two code segments are equivalent, the first one is much easier to read:

```
// Normal "else if" syntax
if (x > y) {
  trace("x is larger than y");
} else if (x < y) {
  trace("x is smaller than y");
} else {
  trace("x and y are equal");
}
// Expanded Nested if/else chain
if (x > y) {
```

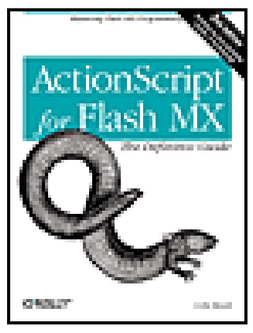
downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

```
  trace("x is larger than y");
} else {
  if (x < y) {
    trace("x is smaller than y");
  } else {
    trace("x and y are equal");
  }
}
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir



ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Chapter 7. Conditionals

7.4 The switch Statement

The *switch* statement, introduced in Flash Player 6, lets us execute one (or sometimes more) of several possible code blocks, based on the value of a single test expression. In some cases, the *switch* statement is easier to write and more legible than an *if* statement. For example, in the following *switch* statement, we greet the user with a custom message depending on the value of the test expression `gender`:

```
var surname = "Porter";
var gender = "male";
switch (gender) {
  case "femaleMarried":
    trace("Hello Mrs. " + surname);
    break;
  case "femaleGeneric":
    trace("Hello Ms. " + surname);
    break;
  case "male":
    trace("Hello Mr. " + surname);
    break;
  default:
    trace("Hello " + surname);
}
```

Our *switch* statement attempts to match the value of `gender` to one of the supplied *case* expressions: "femaleMarried", "femaleGeneric", or "male". Because `gender` matches the expression "male", the substatement *trace("Hello Mr. " + surname);* is executed. If the test expression had not matched any *case* clause, then the default statement—*trace("Hello" + surname);*—would have been executed.

The general form of the *switch* statement is:

```
switch (testExpression) {
  case expression1:
    substatements1
    break;
  case expression2:
    substatements2
    break;
  default:
    substatements3
    break;
}
```

where *`testExpression`* is a value that the interpreter will attempt to match with each of the supplied *case* expressions, from top to bottom. The *case* expressions are supplied with the statement label *case*, followed by a colon.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

If *testExpression* matches a *case* expression, all statements immediately following that *case* label are executed, including those in any subsequent *case* blocks! To prevent subsequent blocks from executing, we must use the *break* statement at the end of each block. If no *case* expression matches *testExpression*, all statements following the *default* label are executed.

Though the *default* label is normally listed last, it can legally come anywhere within the *switch* statement. Furthermore, the *default* label is not mandatory in a *switch* statement. If no *default* is provided and *testExpression* does not match a *case* expression, execution flow simply continues after the end of the *switch* statement block (the code within the *switch* statement is skipped). However, if both the *case* and the *default* labels are omitted, the *switch* statement generates an error (unlike JavaScript, which permits empty *switch* statements).

To provide *switch* with value ranges rather than a single value, use the following technique:

```
switch (true) {
  case (x < 10):
    trace("x is less than 10.");
    break;
  case (x >= 10 && x <= 20):
    trace("x is between 10 and 20.");
    break;
  case (x > 20):
    trace("x is greater than 20.");
    break;
}
```

In this example, note that if more than one *case* yields `true`, the code following the first true expression is executed.

To test for a group of cases rather than a single case, use the following "fall-through" technique (notice how the *break* statement is intentionally omitted from *case 1* and *case 3*:

```
x = 3;

switch (x) {
  case 1:
  case 3:
  case 5:
    trace("x is 1, 3, or 5");
    break;
  default:
    trace("x is not 1, 3, or 5");
}
```

The *switch* statement implicitly uses the strict equality operator (`= = =`), not the equality operator (`= =`), when comparing the *testExpression* with *case* expressions. For a description of the difference, see Chapter 5. The expressions following the `switch` and `case` keywords can be any datatype and need not be literal values. Because a *switch* statement executes from top to bottom, *case* expressions that are most likely to match *testExpression* should be listed first (obviously, this applies only to situations where *case* statement order does not affect the result of the *switch* statement).

Although the *switch* statement was introduced for Flash Player 6, code using *switch* can be exported from the Flash MX authoring tool to Flash Player 5 and Flash Player 4 formatted movies. When exporting to Flash 4 or Flash 5 format, however, *switch* expression matching uses the standard equality (`= =`), not the strict equality (`= = =`) operator. In any case, *switch* does not offer any performance improvement over the analogous *if-else if-else*

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

construction.

Note that `case` and `default` clauses are two specific examples of a more general structure called a *statement label*. Statement labels are a part of ECMAScript v3, but—other than `case` and `default`—they are not supported by ActionScript.

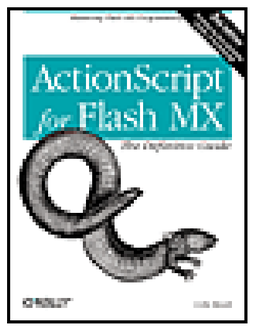
7.4.1 Simulating the switch Statement in Flash 5

Though the *switch* statement is not supported in the Flash 5 authoring tool, it can be emulated using a chain of *if-else* *if-else* statements, like this:

```
var surname = "Porter";
var gender = "male";
if (gender =  = "femaleMarried") {
  trace("Hello Mrs. " + surname);
} else if (gender =  = "femaleGeneric") {
  trace("Hello Ms. " + surname);
} else if (gender =  = "male") {
  trace("Hello Mr. " + surname);
} else {
  trace("Hello " + surname);
}
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir



ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Chapter 7. Conditionals

7.5 Compact Conditional Syntax

Throughout this book we use statement blocks (statements surrounded by curly braces) with all conditional statements, even if a block contains only one statement:

```
if (x =  = y) {
  trace("x and y are equal");
}
```

For the sake of brevity, ActionScript does not require curly braces when a conditional has only one substatement. A single substatement can, quite legitimately, be placed directly after an *if* or *else if* statement without any curly braces, like this:

```
if (x =  = y) trace ("x and y are equal");
```

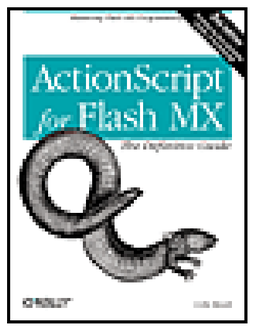
or even on the next line, like this:

```
if (x =  = y)
  trace ("x and y are equal");
```

For some programmers, this style can be a little slower to read and slightly more error-prone, though it undeniably saves room in source code.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir



ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Chapter 7. Conditionals

7.6 Onward!

Conditional statements are an immensely important aspect of ActionScript. They give us precise control over the execution of our code. In the next chapter, we'll examine another kind of important execution-control statement, *loops*.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Part I: ActionScript Fundamentals

Chapter 8. Loop Statements

In Chapter 7, we saw that a conditional causes a statement block to be executed once if the value of its test expression is `true`. A *loop*, on the other hand, causes a statement block to be executed repeatedly, for as long as its test expression remains `true`.

Loops come in a variety of tasty flavors: *while*, *do-while*, *for*, and *for-in*. The first three types have very similar effects, but with varying syntax. The last type of loop, *for-in*, is a specialized kind of loop used with objects. We'll start our exploration of loops with the *while* statement, the easiest kind of loop to understand.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Chapter 8. Loop Statements

8.1 The while Loop

Structurally, a *while* statement is constructed much like an *if* statement: a main statement encloses a block of substatements th are executed only when a given condition is `true` :

```
while (condition) {
  substatements
}
```

If `condition` is `true` , `substatements` are executed. But, unlike the *if* statement, when the last substatement is finished, execution begins anew at the beginning of the *while* statement (that is, the interpreter "loops" back to the beginning the *while* statement). The second pass through the *while* statement works just like the first: the condition is evaluated, and if i still `true` , `substatements` are executed again. This process continues until `condition` becomes `false` , at whic point execution continues with any statements that follow the *while* statement in the script.

Here's an example of a very simple loop:

```
var i = 3;
while (i < 5) {
  trace("x is less than 5");
}
```

The example reliably represents the correct syntax of a *while* loop but is most likely in error. To see why, let's follow along w the interpreter as it executes the example.

The interpreter starts with the statement before the *while* statement, *var i = 3* , which sets the variable `i` to 3. Because the variable `i` is used in the test expression of the loop, this step is often called the *loop initialization* . Next, the interpreter begin executing the *while* statement by resolving the test expression: *i < 5* . Because `i` is 3, and 3 is less than 5, the value of the test expression is `true` ; so, the interpreter executes the *trace( )* statement within the loop, sometimes called the *loop body* .

With that done, it's time to repeat the loop. Once again, the interpreter checks the value of the test expression. The value of th variable `i` has not changed, so the test expression is still `true` and the interpreter executes the *trace( )* statement again. At th point, the loop body has finished executing, so it's time to repeat the loop. Guess what? The variable `i` still has not changed, the test expression is still `true` and the interpreter must execute the *trace( )* statement again, and again, and again, forever. Because the test expression always returns `true` , there's no way to exit the loop—Flash is trapped forever in an *infinite loop* In ActionScript, an infinite loop causes an error, as we'll see later.

Our loop is infinite because it lacks an *update statement* that changes the value of the variable used in the test expression. An update statement typically causes the test expression to eventually yield `false` , which terminates the loop. Let's fix our infi loop by adding an update statement:

```
var i = 3;
while (i < 5) {
  trace("x is less than 5");
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

```
  i++;
}
```

The update statement, *i++;* , comes at the end of the loop body. When the interpreter goes through our loop, it executes the *tra* *)* statement as before, but it also executes the statement *i++;* , which adds 1 to the variable `i` . With each iteration of the loop value of `i` increases. After the second iteration, `i` 's value is 5, so the test expression, *i < 5* , becomes `false` . The loop, therefore, safely ends after two iterations.

Our loop's update statement performs a fundamental loop activity: it counts. The variable `i` (called a *counter*) runs through a predictable numeric sequence—perfect for methodical tasks such as duplicating movie clips or accessing the elements of an array. Here we duplicate the `square` movie clip five times without using a loop:

```
// Name each new clip sequentially and place it on its own depth
duplicateMovieClip("square", "square1", 1);
duplicateMovieClip("square", "square2", 2);
duplicateMovieClip("square", "square3", 3);
duplicateMovieClip("square", "square4", 4);
duplicateMovieClip("square", "square5", 5);
```

And here we perform the same operation with a loop:

```
var i = 1;
while (i <= 5) {
  duplicateMovieClip("square", "square" + i, i);
  i++;
}
```

Imagine the difference if we were duplicating `square` 100 times!

Loops are marvelously useful for manipulating data, particularly data stored in arrays. Example 8-1 shows a loop that displays all the elements of an array to the Output window. Note that the first element is number 0, not number 1.

Example 8-1. Displaying an array with a while loop

```
var people = ["John", "Joyce", "Margaret", "Michael"];  // Create an array
var i = 0;
while (i < people.length) {
  trace("people element " + i + " is " + people[i]);
  i++;
}
```

The result in the Output window is:

```
people element 0 is John
people element 1 is Joyce
people element 2 is Margaret
people element 3 is Michael
```

Notice that the variable `i` is used both in the test expression and as the array index number, as is typical. Here we use `i` again an argument for the *charAt( )* function:

```
var city = "Toronto";
trace("The letters in the variable 'city' are ");
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

```
var i = 0;
while (i < city.length) {
  trace(city.charAt(i));
  i++;
}
```

The Output window shows:

```
The letters in the variable 'city' are:
T
o
r
o
n
t
o
```

Finally, instead of dissecting data, we use a loop to construct a sentence from a series of words stored in an array:

```
var words = ["Toronto", "is", "not", "the", "capital", "of", "Canada"];
var sentence;
var i = 0;
while (i < words.length) {
  sentence += words[i];       // Add the current word to the sentence
  // If it's not the last word...
  if (i < words.length - 1) {
    sentence += " ";          // ...tack on a space
  } else {
    sentence += ".";          // ...otherwise, end with a period
  }
  i++;
}
trace(sentence);              // Displays: "Toronto is not the capital of Canad
```

Nearly all loops involve some kind of counter (sometimes called an *iterator* or *index variable*). Counters let us cycle sequenti through data. This technique is particularly convenient when we determine the counter's maximum limit using the `length` property of the array or string we want to manipulate, as we did in the preceding example.

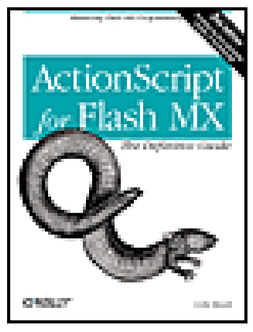
It's also possible to create a loop whose end point doesn't depend on a counter. As long as the test expression of the loop eventually becomes `false` , the loop will end. Here, for example, we examine the document level stack of the Flash Player t determine the first vacant level:

```
var i = 0;
while (typeof eval("_level" + i) =  = "movieclip") {
  i++;
}
trace("The first vacant level is " + i);
// Now load a movie into the vacant level, knowing it's free
loadMovieNum("myMovie.swf", i);
```

For more information on document levels, see Section 13.4 in Chapter 13.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir



ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Chapter 8. Loop Statements

8.2 Loop Terminology

In the previous section we introduced several new terms. Let's look at these more formally, so that you'll understand them well when working with loops:

Initialization

The statement or expression that defines one or more variables used in the test expression of a loop.

Test expression

The condition that must be met in order for the substatements in the loop body to be executed (often called a *condition* or *test*, or sometimes, *control*).

Update

The statements that modify the variables used in the test expression before a subsequent test. A typical update statement increments or decrements the loop's counter.

Iteration

One complete execution of the test expression and statements in the loop body (sometimes referred to as one *loop* or one *pass*).

Nesting or nested loop

A loop that contains another loop, allowing you to iterate through some sort of two-dimensional data. For example, you might loop through each row in a column for all the columns in a table. The outer or top-level loop would progress through the columns, and the inner loop would progress through the rows in each column.

Iterator or index variable

A variable whose value increases or decreases with each iteration of a loop, usually used to count or sequence through some data. Loop iterators are often called *counters*. Iterators are conventionally named `i`, `j`, and `k`, or sometimes `x`, `y`, and `z`. In a series of nested loops, `i` is usually the iterator of the top-level loop, `j` is the iterator of the first nested loop, `k` is the iterator of the second nested loop, and so on. You can use any variable name you like for clarity. For example, you can use `charNum` as the variable name to remind yourself that it indicates the current character in a string.

Loop body

The block of statements that is executed when a loop's condition is met. The body may not be executed at all, or it may be executed thousands of times.

Loop header

The portion of a loop that contains the loop statement keyword (*while*, *for*, *do-while*, or *for-in*) and the loop controls. The loop control varies with the type of loop. In a *for* loop, the control comprises the initialization, the test, and the update; in a *while* loop, the control comprises the test expression only.

Infinite loop

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

A loop that repeats forever because its test expression never yields the value `false`. Infinite loops cause an error in ActionScript, as discussed later in this chapter under Section 8.6.3.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Chapter 8. Loop Statements

8.3 The do-while Loop

As we saw earlier, a *while* statement allows the interpreter to execute a block of code repeatedly while a specified condition remains `true`. Due to a *while* loop's structure, its body will be skipped entirely if the loop's condition is not met the first time it is tested. A *do-while* statement lets us guarantee that a loop body will be executed at least once with minimal fuss. The body of a *do-while* loop always executes the first time through the loop. The *do-while* statement's syntax is somewhat like an inverted *while* statement:

```
do {
  substatements
} while (condition);
```

The keyword `do` begins the loop, followed by the *`substatements`* of the body. On the interpreter's first pass through the *do-while* loop, *`substatements`* are executed before *`condition`* is ever checked. At the end of the *`substatements`* block, if *`condition`* is `true`, the loop is begun anew and *`substatements`* are executed again. The loop executes repeatedly until *`condition`* is `false`, at which point the *do-while* statement ends. Note that a semicolon is required following the parentheses that contain the *`condition`*.

Obviously, *do-while* is handy when we want to perform a task at least once and perhaps subsequent times. In Example 8-2 we duplicate a series of twinkling-star movie clips from a clip called `starParent` and place them randomly on the Stage. Our galaxy will always contain at least one star, even if `numStars` is set to 0.

Example 8-2. Using a do-while loop

```
var numStars = 5;
var i = 1;
do {
  // Duplicate the starParent clip
  duplicateMovieClip(starParent, "star" + i, i);
  // Place the duplicated clip randomly on stage
  _root["star" + i]._x = Math.floor(Math.random() * 551);
  _root["star" + i]._y = Math.floor(Math.random() * 401);
} while (i++ < numStars);
```

Did you notice that we sneakily updated the variable `i` in the test expression? Remember from Chapter 5 that the post-increment operator both returns the value of its operand and also adds 1 to that operand. The increment operator is very convenient (and common) when working with loops.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Chapter 8. Loop Statements

8.4 The for Loop

A *for* loop is essentially synonymous with a *while* loop, but it is written with more compact syntax. Most notably, the loop header can contain both initialization and update statements in addition to the test expression.

Here's the syntax of the *for* loop:

```
for (initialization; condition; update) {
  substatements
}
```

The *for* loop places the key components of a loop tidily in the loop header, separated by semicolons. Before the first iteration of a *for* loop, the `initialization` statement is performed (once and only once). It is typically used to set the initial value of one or more iterator variables. As with other loops, if `condition` is `true`, `substatements` are executed. Otherwise, the loop ends. Even though it appears in the loop header, the `update` statement is executed at the *end* of each loop iteration, before `condition` is tested again to see if the loop should continue. Here's a typical *for* loop that simply counts from 1 to 10:

```
for (var i = 1; i <= 10; i++) {
  trace("Now serving number " + i);
}
```

It's easier to understand how a *for* loop works when you see its equivalent constructed using the *while* loop syntax:

```
var i = 1;
while (i <= 10) {
  trace("Now serving number " + i);
  i++;
}
```

Once you're used to the *for* syntax, you'll find it saves space and allows for easy interpretation of the loop's body and controls.

8.4.1 Multiple Iterators in for Loops

If we want to control more than one factor in a loop, we can use more than one iterator variable. A *while* loop with multiple iterators might look like this:

```
var i = 1;
var j = 10;
while (i <= 10) {
  trace("Going up " + i);
  trace("Going down " + j);
  i++;
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

```
  j--;
}
```

The same effect can be achieved in a *for* statement using the comma operator:

```
for (var i = 1, j = 10; i <= 10; i++, j--) {
  trace("Going up " + i);
  trace("Going down " + j);
}
```

Notice that the *var* keyword is listed only once—it applies to all supplied variables.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Chapter 8. Loop Statements

8.5 The for-in Loop

A *for-in* statement is a specialized loop used to list the properties of an object. New programmers may want to skip this section for now and return to it after reading Chapter 12.

Unlike other loops, which repeat a series of statements until a given test expression is `false`, a *for-in* loop iterates once for each property in the specified object. Therefore, *for-in* statements do not need an explicit update statement because the number of loop iterations is determined by the number of properties in the object being inspected. The syntax of a *for-in* loop looks like this:

```
for (var thisProp in object) {
  substatements  // Substatements typically use thisProp in some way
}
```

The *`substatements`* are executed once for each property of *`object`*; *`object`* is a reference to any valid object; *`thisProp`* is a variable name (and should not be hardcoded with the name of an existing object property). During each loop iteration, the *`thisProp`* variable is automatically set to a string that is the name of the object property currently being enumerated. That string value can be used during each iteration to access and manipulate the current property. The simplest example of a *for-in* loop is one that lists the properties of an object. Here we create an object and then enumerate its properties with a *for-in* loop:

```
var ball = new Object( );
ball.radius = 12;
ball.color = "red";
ball.style = "beach";
for (var prop in ball) {
  trace("ball has the property " + prop);
}
```

Because `prop` stores the names of the properties of `ball` as strings, we can use `prop` with the `[ ]` operator to retrieve the values of those properties, like this:

```
for (var prop in ball) {
  trace("ball." + prop + " is " + ball[prop]);
}
```

Retrieving property values with a *for-in* loop also provides a super way to detect the movie clips present on a timeline. For a demonstration of the *for-in* loop used as a movie clip detector, see Example 3-1.

Note that the properties of the object being inspected in a *for-in* loop are not enumerated in any predictable order. Also, *for-in* statements do not always list every property of an object. When the object is user-defined, all properties are enumerated, including any inherited properties. But some properties of built-in objects are skipped by the *for-in* statement. Methods of built-in objects, for example, are not enumerated by a *for-in* loop. If you want to use a *for-in* statement to manipulate the properties of a built-in object, first build a test loop to determine the object's accessible

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

properties, such as:

```
thisFormat = new TextFormat( );
for (var prop in thisFormat) {
  trace("Property found: " + prop);
}
```

When the *object* being inspected is an array, the value of *thisProp* is a string containing the integer index of the element being enumerated.

To prevent a user-defined object property from appearing in a *for-in* loop, you can use the undocumented *ASSetPropFlags( )* function as follows:

```
// Create a new ball object with radius and color properties.
var ball = new Object( );
ball.radius = 10;
ball.color = "red";

// Hide ball.radius from for-in loops.
ASSetPropFlags(ball, ["radius"], 1);

// Displays: color (radius is hidden)
for (var prop in ball) {
  trace(prop);
}
```

This technique is not officially documented by Macromedia and may be replaced by a more intuitive function in the future. It is available in Flash Player 5 and Flash Player 6. For more information, see:

http://chattyfig.figleaf.com/flashcoders-wiki/index.php?ASSetPropFlags

In Flash Player 5, input text fields without a default value are not enumerated by a *for-in* loop. Hence, form-validation code that detects empty text fields will not work properly unless those text fields are explicitly declared as variables in the timeline upon which they reside. In Flash Player 6, all text fields are represented by objects and are enumerated by *for-in* even when empty.

The *for-in* statement can also be used to extract elements in an array, in which case it takes the form:

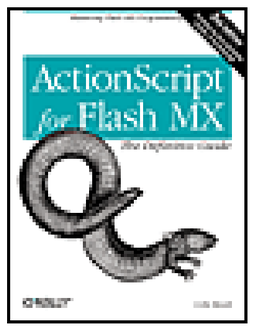
```
for (var thisElem in array) {
  substatements; // Substatements typically use thisElem in some way
}
```

This example lists the elements of an array:

```
var myArr = [123, 234, 345, 456];
for (var elem in myArr) {
  trace(myArr[elem]);
}
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir



ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Chapter 8. Loop Statements

8.6 Stopping a Loop Prematurely

In a simple loop, the test expression is the sole factor that determines when the loop stops. When the test expression of a simple loop yields `false`, the loop terminates. However, as loops become more complex, we may need to terminate a running loop arbitrarily, regardless of the value of the test expression using the *break* statement. A related statement, *continue*, ends the current iteration of the loop body and resumes execution with the next iteration.

8.6.1 The break Statement

The *break* statement ends execution of the current loop or *switch* statement (as we saw in Chapter 7). It has the modest syntax:

```
break
```

The only requirement is that *break* must appear within the body of a loop or *switch* statement.

The *break* statement provides a way to halt a process that is no longer worth completing. For example, we can use a *for-in* loop to build a form-checking routine that cycles through the input-text variables on a timeline. If a blank input field is found, we alert the user that she hasn't filled in the form properly. We can abort the process by executing a *break* statement. Example 8-3 shows the code. Note that the example assumes the existence of a movie clip called `form_mc` that contains a series of declared input variables named `input01_txt`, `input02_txt`, and so on.

Example 8-3. A simple form-field validator

```
for (var prop in form_mc) {
  // If this property is one of our "input" text fields
  if (prop.indexOf("input") != -1) {
    // If the form entry is blank, abort the operation
    if (form[prop].text =  = "") {
      displayMessage = "Please complete the entire form.";
      break;
    }
    // Any substatements following the break command are not reached
    // when the break is executed
  }
}
// Execution resumes here after the loop terminates whether
// due to the break command or the test condition becoming false
```

8.6.2 The continue Statement

The *continue* statement is similar to the *break* statement in that it causes the current iteration of a loop to be aborted;

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

but, unlike *break*, it resumes the loop's execution with the next natural cycle. The syntax of the *continue* statement is simply:

```
continue
```

In all types of loops, the *continue* statement interrupts the current iteration of the loop body, but the resumption of the loop varies slightly depending on the type of loop statement. In a *while* loop and a *do-while* loop, the test expression is checked before the loop resumes. But in a *for* loop, the loop update is performed before the test expression is checked. And in a *for-in* loop, the next iteration begins with the next property of the object (or element of the array), if one exists.

Using the *continue* statement, we can make the execution of the body of a loop optional under specified circumstances. For example, here we move all the movie clip instances that aren't transparent to the left edge of the Stage, and we skip the loop body for transparent instances:

```
for (var prop in _root) {
  if (typeof _root[prop] =  = "movieclip") {
    if (_root[prop]._alpha < 100) {
      continue;
    }
    _root[prop]._x = 0;
  }
}
```

8.6.3 Maximum Number of Iterations

As noted earlier in this chapter, loops are not allowed to execute forever in ActionScript. In Flash Player 5 or later, loops are limited to 15 seconds. The number of iterations that can be achieved in that time depends on what's inside the loop and the computer's speed. To be safe, you shouldn't create loops requiring more than even a few seconds to execute (which is eons in processing terms!). Most loops should take only milliseconds to finish. If a loop takes longer to complete (for example, because it's processing hundreds of strings while initializing a word-scramble game), it's worth rewriting the code using a timeline loop, as described in Section 8.7. Timeline loops allow us to update the progress of a script's execution on screen and avoid the potential display of the error message shown in Figure 8-1.

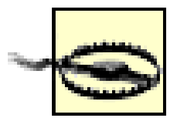
Figure 8-1. Bad loop! Down boy!

When a loop has run for more than 15 seconds in Flash Player 5 or later, an alert box warns the user that a script in the movie is delaying the movie's playback. The user is offered the choice to either wait for the script to finish or to quit the script.

Flash Player 4 is even stricter—it allows only 200,000 iterations, after which all scripts are disabled without any warning.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir



Take special heed: the 15-second warning that users see does not mention that canceling a runaway script will actually cause all scripts in the movie to stop functioning! If a user selects Yes to stop a loop from continuing, all scripts in the movie are disabled.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Chapter 8. Loop Statements

8.7 Timeline and Clip Event Loops

The loops we've looked at so far cause the interpreter to execute blocks of code repeatedly. Most of your loops will be of this "ActionScript-statement" type. Sometimes it's desirable to create a *timeline loop* by looping Flash's playhead in the timeline. To do so, attach a series of statements to any frame; on the next frame, attach a *gotoAndPlay( )* function whose destination is the previous frame. When the movie plays, the playhead will cycle between the two frames, causing the code on the first frame to execute repeatedly.

We can make a simple timeline loop by following these steps:

1. Create a new Flash movie.
2. On frame 1, attach the following statement:

```
trace("Hi there! Welcome to frame 1");
```

3. On frame 2, add a blank keyframe and attach the following statements:

```
trace("This is frame 2");
this.gotoAndPlay(1);
```

4. Select Control Test Movie.

When we test our movie, we see an endless stream of the following text:

```
Hi there! Welcome to frame 1
This is frame 2
Hi there! Welcome to frame 1
This is frame 2
```

Timeline loops can do two things ordinary loops cannot:

- They can execute a block of code an infinite number of times without causing an error.
- They can execute a block of code that requires a Stage update between loop iterations.

This second feature of timeline loops requires a little more explanation. When any frame's script is executed, the movie Stage is not updated visually until the end of the script. This means that traditional loop statements cannot be used to perform repetitive visual or audio tasks because the task results aren't rendered between each loop iteration. Repositioning a movie clip, for example, requires a Stage update, so we can't programmatically animate a movie clip using a normal loop statement.

You might assume that the following code would visually slide the `ball_mc` movie clip horizontally across the Stage:

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

```
for (var i = 0; i < 50; i++) {
  ball_mc._x += 10;
}
```

Conceptually, the loop statement has the right approach—it repetitively updates the position of `ball_mc` by small amounts, which should give the illusion of movement. However, in practice, `ball_mc` doesn't move each time its `_x` position is changed, because the Stage isn't updated between loop iterations. Instead, we'd see `ball_mc` suddenly jump 500 pixels to the right—10 pixels for each of the 50 loop iterations—after the script completes.

To allow the Stage to update after each execution of the `ball_mc._x += 10;` statement, we can use a timeline loop like this:

```
// CODE ON FRAME 1
ball_mc._x += 10;
// CODE ON FRAME 2
this.gotoAndPlay(1);
```

Because Flash updates the Stage between any two frames, the `ball` will appear to animate. But the timeline loop completely monopolizes the timeline it's on. While it's running, we can't animate any other content on that timeline. A better approach is to put our timeline loop into an empty, two-frame movie clip. We'll get the benefit of a Stage update between loop iterations without freezing a timeline we may need for other animation.

8.7.1 Creating an Empty-Clip Timeline Loop

The following steps show how to create an empty-clip timeline loop:

1. Create a new Flash movie.
2. Create a movie clip symbol, named *ball*, that contains a circle shape.
3. On the main Stage, rename layer *Layer 1*, *ball*.
4. On the *ball* layer, place an instance of the *ball* symbol.
5. Name the instance of the *ball* clip `ball_mc`.
6. Select Insert New Symbol to create a blank movie clip symbol.
7. Name the clip symbol *process*.
8. On frame 1 of the *process* symbol, attach the following code:

```
_root.ball_mc._x += 10;
```

9. On frame 2 of the *process* symbol, add a blank keyframe and attach the following code:

```
this.gotoAndPlay(1);
```

10. Return to the main movie timeline and create a layer called *scripts*.
11. On the *scripts* layer, place an instance of the *process* symbol.
12. Name the instance `processMoveBall_mc`.
13. Select Control Test Movie.

The `processMoveBall_mc` instance will now move `ball_mc` without interfering with the playback of the main timeline upon which `ball_mc` resides.

Note that step 12 isn't mandatory, but it gives us more control over our loop. By giving our *process* instance a name, we can stop and start our loop by starting and stopping the playback of the instance, like this:

```
processMoveBall_mc.play( );
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

```
processMoveBall_mc.stop( );
```

Note that in this example, `processMoveBall_mc` and `ball_mc` must both exist on the main timeline for as long as the loop is supposed to work. If we want to make the code more portable, we can use a relative reference to our `ball_mc` clip in the *process* symbol:

```
_parent.ball_mc._x += 10;
```

Timeline loops can't loop on a single frame. That is, if we place a *gotoAndPlay(5)* function on frame 5 of a movie, the function will be ignored. The Flash Player realizes that the playhead is already on frame 5, and it simply does nothing.

You'll find the sample timeline loop and empty-clip loop *.fla* files in the online Code Depot.

8.7.2 Movie Clip onEnterFrame( ) Loops

Timeline loops are effective but not necessarily elegant. We can use an event handler on a movie clip to achieve the same results as a timeline loop but with more flexibility (just try to follow along with this example, or see Chapter 10 for details on movie clip event handlers).

When placed on a movie clip, an *onEnterFrame( )* event handler causes a block of code to execute once per tick of the frame rate of a movie. We can use an *onEnterFrame( )* event handler on a single-frame empty clip to execute a block of code repetitively, while allowing for a Stage update between each repetition (just as a timeline loop does). Follow these steps to try it out:

1. Create a `ball_mc` instance by following steps 1 through 5 from the previous section.
2. On the main timeline, create a new layer called *scripts*.
3. On frame 1 of the *scripts* layer, attach the following code:

```
// This code requires Flash Player 6 or later
this.createEmptyMovieClip("processMoveBall_mc", 1);
processMoveBall_mc.onEnterFrame = function ( ) {
  _root.ball_mc._x += 10;
}
```

4. Select Control Test Movie.

The `ball_mc` instance should animate across the Stage.

The *onEnterFrame( )* loop frees us from nesting our code inside a movie clip and doesn't require a two-frame loop, as timeline loops do. All the action of a clip event loop happens in a single event handler. To stop an *onEnterFrame( )* loop, simply remove the movie clip that bears it. For example:

```
processMoveBall_mc.removeMovieClip( );
```

Or, delete the *onEnterFrame( )* handler:

```
delete processMoveBall_mc.onEnterFrame;
```

If our application is simple, we may wish to forego our empty clip *onEnterFrame( )* loop altogether. In some cases, we can, quite legitimately, attach an event loop directly to the clip being manipulated. In our `ball_mc` example, we can avoid the need for a separate empty clip by defining our *onEnterFrame( )* handler directly on the `ball_mc` instance:

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

```
ball_mc.onEnterFrame = function ( ) {
  this._x += 10;
}
```

We can even define the *onEnterFrame( )* event loop on the main timeline by placing this code on one of its frames:

```
this.onEnterFrame = function ( ) {
  this.ball_mc._x += 10;
}
```

Notice that we intentionally avoid referring to the main timeline as `_root`, which may change meaning if the entire movie is loaded into a movie clip.

In Flash Player 6, each movie clip (including the main timeline) can take only one *onEnterFrame( )* event handler. In most applications, a single, main application loop runs once per frame, updating the entire movie. For example:

```
// CODE ON MAIN TIMELINE
function mainLoop ( ) {
  // Perform application-wide updates here
}
this.onEnterFrame = mainLoop;
```

Less frequently, multiple movie clips can update various aspects of the application, each with their own *onEnterFrame( )* handler.

8.7.3 Frame Rate's Effect on Timeline and onEnterFrame( ) Loops

Because timeline and *onEnterFrame( )* loops iterate once per frame, their execution frequency is tied to the frame rate of a movie. If we're moving an object around the screen with a timeline or an event loop, an increase in frame rate can mean an increase in the speed of our animation.

When we programmed the movement of the `ball_mc` clip in our earlier examples, we implicitly specified the velocity of the ball in relation to the frame rate. Our code says, "With each frame that passes, move `ball_mc` ten pixels to the right":

```
ball_mc._x += 10;
```

Hence, `ball_mc`'s velocity is dependent on the frame rate. If our movie plays at 12 frames per second, then our `ball_mc` clip moves 120 pixels per second. If our movie plays at 30 frames per second, our `ball_mc` clip moves *300* pixels per second!

When timing scripted animations, it's tempting to calculate the distance to move an item in relation to the movie's frame rate. So, if a movie plays 20 frames per second, and we want an item to move 100 pixels per second, we're tempted to set the speed of the object to 5 pixels per frame (5 pixels * 20 frames per second = 100 pixels per second). There are two serious flaws in this approach:

- By relying on the frame rate to determine the speed of an item, we make it painful to change the frame rate. If we change the frame rate, we have to recalculate our speed and edit our code accordingly.
- The Flash Player does not necessarily play movies back at the frame rate set in the Flash authoring tool; it often plays them slower. If the computer running the movie cannot render frames fast enough to keep up with the designated frame rate, the movie slows down. This slowdown can even vary depending on the system load; if

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

other programs are running or if Flash is performing some processor-intensive task, the frame rate may drop for only a short period and then resume its normal pace.

You can test a movie's actual frame rate yourself using the time-tracker tool available at:

http://www.moock.org/webdesign/flash/actionscript/fps-speedometer

In some cases, an animation that plays back at slightly different speeds can be deemed acceptable. But when visual accuracy matters or when we're concerned with the responsiveness of an action game, it's much more appropriate to calculate the distance to move an object based on elapsed time rather than relative to the frame rate. Example 8-4 shows a quick-and-dirty sample of time-based animation (i.e., the `ball_mc` speed is independent of the frame rate). The new movie has three frames and two layers, one layer with the `ball_mc` instance and the other with our scripts.

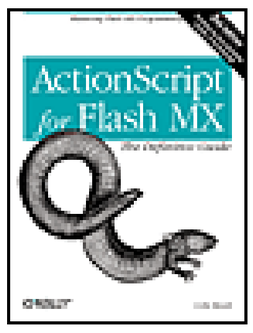
Example 8-4. Calculating move distances based on time, not frame rate

```
// CODE ON FRAME 1
var distancePerSecond = 50;  // Pixels to move per second
var now = getTimer();        // The current time
var then = 0;                // The time when last frame was rendered
var elapsed;                 // Milliseconds between frame renders
var numSeconds;              // elapsed expressed in seconds
var moveAmount;              // Distance to move each frame

// CODE ON FRAME 2
then = now;
now = getTimer();
elapsed = now - then;
numSeconds = elapsed / 1000;
moveAmount = distancePerSecond * numSeconds;
ball_mc._x += moveAmount;

// CODE ON FRAME 3
gotoAndPlay(2);
```

Note that our time-based movement might appear jerky if the frame rate suddenly changes. We could smooth things out by using an elapsed-time measurement that averages the time between a series of frames instead of just two.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir



ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Chapter 8. Loop Statements

8.8 An Alternative to Timeline Loops:setInterval( )

As of Flash Player 6, we can use *setInterval( )* to execute a function every *n* milliseconds. For example, the following code creates a function, *moveBall( )*, that moves the `ball_mc` movie clip 10 pixels to the right, and then uses *setInterval( )* to execute that function every 20 milliseconds.

```
moveBall = function ( ) {
  _root.ball_mc._x += 10;
}
ballMoverID = setInterval(moveBall, 20);
```

To stop our interval, we can later use:

```
clearInterval(ballMoverID);
```

Like a timeline loop or an *onEnterFrame( )* loop, *setInterval( )* can execute a block of code indefinitely, and it allows for screen updates between iterations. However, the *setInterval( )* function is not tied directly to a movie's frame rate (though it is affected by it). For complete details, see *setInterval( )* in the ActionScript Language Reference.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Chapter 8. Loop Statements

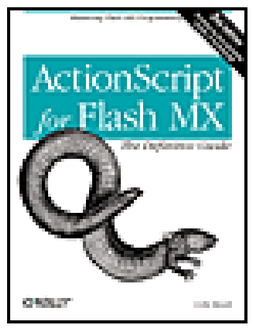
8.9 Onward!

Well, we've come pretty far. The end of this chapter marks a milestone—it ends our examination of the ActionScript statements. This means we have variables, data, datatypes, expressions, operators, and statements under our belt. These elements of the language are the foundation of all scripts. If you've read and understood everything up to this point, or at least most of it, you can officially claim that you're able to "speak" ActionScript.

In the remainder of Part I, we'll work on making our conversations more eloquent and our commands more powerful. We'll consider the advanced topics of how to make code portable, how to create events that initiate the execution of our code, how to manage complex data, and how to manipulate movie clips programmatically. These techniques will help us build more advanced applied examples.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir



ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Part I: ActionScript Fundamentals

Chapter 9. Functions

I'm almost giddy to tell you about functions because they're such a powerful part of ActionScript. A function is simply a chunk of code that can be reused throughout a program. Not only do functions lend enormous flexibility and convenience to our scripts, they also give us control over Flash movie elements. I can hardly imagine programming without functions—they ease performing common operations, from sorting words to calculating the distance between two movie clips. We'll introduce functions in this chapter before learning how to create complex, powerful programs using functions with objects in Chapter 12.

We'll focus first on the simplest functions to understand—*custom functions*—the functions we create in our scripts, as distinguished from the *built-in functions* provided with ActionScript.

The ECMA-262 specification refers to custom functions as *program functions* and built-in functions as *internal functions*. We use the colloquial name in both cases rather than the formal specification's terminology.

By learning to create your own functions, you'll become familiar with these fundamentals:

Function declaration or function definition

Creating functions for future use in our scripts.

Function invocation

Causing functions to execute. In other words, running the code in a function. Also known as *calling a function*.

Function arguments and parameters

Providing functions with data to manipulate upon invocation.

Function termination

Ending the execution of a function and, optionally, returning a result.

Function scope

Determining the availability and life span of a function, and the accessibility of variables referenced in a function body.

Let's get to it!

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Chapter 9. Functions

9.1 Creating Functions

To make a basic *function* we specify a function name and a block of statements to perform, like this:

```
function funcName (param1, param2,...paramn) {
  statements
}
```

The `function` keyword starts the *declaration* of our new function. Next comes our function name, `funcName`, which we'll use later to *invoke* our function. The specified `funcName` must be a legal identifier[1], but otherwise it can be any unique name we choose (avoid creating functions with the same name as built-in functions). Next, we supply a pair of parentheses, `()`, that enclose any parameters (`param1`, `param2`, etc.), separated by commas, which we'll discuss later. If our function does not have any parameters, we include the parentheses but leave them empty. Finally, we provide the *function body* (i.e., statement block), which contains the code that's executed when our function is *called*.

[1] See Chapter 15 for the rules that govern legal identifiers.

Let's make a very simple function:

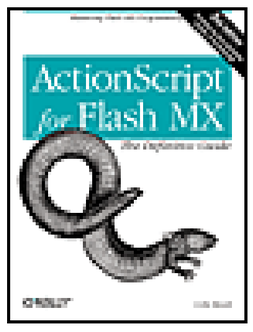
1. Create a new empty Flash movie.
2. On frame 1 of Layer 1 of the main movie timeline, attach the following code:

```
function sayHi () {
  trace("Hi there!");
}
```

That was easy—we've just created a function named *sayHi( )*. This simple function does not require any parameters. When we run it, the *trace( )* statement in its body will be executed. Don't close the movie you've created; you'll learn to run the function next.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir



ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Chapter 9. Functions

9.2 Running Functions

We run a function like a wizard invokes a spell, by proclaiming the function's name, followed by the function call operator, `()`, introduced in Chapter 5:

```
funcName(param1, param2,...paramn)
```

(You can almost detect the faint odor of smoldering mandrake root in the air.)

The parentheses contain the values for any parameters required by the function. If the function requires no parameters (as determined by the function declaration), the parentheses are left empty. Let's try invoking our *sayHi( )* function, which has no parameters, to get the hang of basic function invocation.

Here, again, is our *sayHi( )* function declaration:

```
function sayHi () {
  trace("Hi there!");
}
```

And here's our *sayHi( )* function invocation:

```
sayHi();
```

In this simple demonstration, we add the function invocation to the same script containing the function declaration, but we'll see later that this isn't always the case. When the *sayHi( )* function call is executed, the *sayHi* function is invoked, so its function body runs, causing "Hi there!" to appear in the Output window.

You can see that typing *sayHi( );* is more convenient than typing the whole *trace( )* statement, and the function name, *sayHi*, is a more meaningful description of what our code does. Using thoughtful function names makes our code more readable, almost like human sentences. In this book and in most programming documentation, function names are conventionally listed with the invocation parentheses, as in "the *sayHi( )* function" versus "the *sayHi* function." This convention makes function names easy to spot in a sentence.

Before we continue, notice the semicolon at the end of the function call:

```
sayHi();
```

We add the semicolon because a function call is a complete statement, and good form dictates that all statements should end in a semicolon.

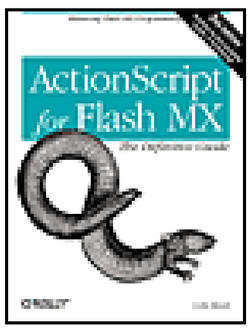
Believe it or not, you've just learned the basics of creating and invoking functions—not too shabby. Functions keep our code centralized and easier to maintain, especially when we need to perform the same operation repeatedly throughout our program. Functions become even more powerful when used with parameters, which we'll consider next.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir



ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Chapter 9. Functions

9.3 Passing Information to Functions

In the last section, we created a function that executed a simple *trace( )* statement—not exactly the most compelling specimen function species. Here's a more interesting function that moves a movie clip instance, named `ball`, a short distance. Follow to create it in your own movie:

1. Create a new empty Flash movie.
2. Create a new movie clip symbol, named *ballSymbol*, that contains a circle shape.
3. Place an instance of *ballSymbol* on Layer 1.
4. Name the *ballSymbol* instance `ball`.
5. On frame 1 of Layer 1 of the main movie timeline, attach the following code:

```
// Store a portable reference to the main timeline.
_global.mainTimeline = this;

function moveBall () {
  mainTimeline.ball._x += 10;
  mainTimeline.ball._y += 10;
}
```

As a matter of good form, functions should not access external timeline variables (such as `ball`) directly. Therefore, we acc through our application's global reference to the movie's main timeline. With the function *moveBall( )* defined, we can move t clip diagonally by calling the *moveBall( )* function. Add the following code to the existing code on frame 1.

```
moveBall();
```

The ball moves diagonally down and to the right. (Note that the origin (0, 0) is in the upper left of the main Stage. Increasing `_x` move the ball to the right, but, unlike the Cartesian coordinates, increasing values of `_y` move the ball *down*, not up.)

Our *moveBall( )* function is convenient, but it lacks flexibility. It works only on one movie clip (`ball`), it moves `ball` in o direction, and it always moves `ball` the same distance.

A well-designed function should work in many circumstances. We can generalize our *moveBall( )* function so that it can mov *any* distance in *any* direction. The first step in generalizing a function is determining what factors control its behavior. In our function, the factors are the name of the movie clip to move, the distance to move it horizontally, and the distance to move it Such factors are known as the *parameters* of the function—they're the information that we'd like to be able to change each tim function is called.

9.3.1 Creating Functions with Parameters

Recall the generic syntax of a simple function declaration:

```
function funcName (param1, param2,...paramn) {
  statements
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

```
}
```

A function's *parameters* are variables that can be used within the function but whose values are determined when the function executed. We specify parameters as a comma-separated list of legal identifiers between the parentheses of a function declarati shown in the previous code fragment.

Once defined, we can access a function's parameter values from inside the function body just as we would any other variable. example, here is a function with one parameter:

```
function say (msg) {                    // Define the msg parameter
  trace("The message is " + msg);  // Use msg within the trace( ) statement
}
```

Our function declaration defines the parameter `msg` . The *trace( )* statement uses the `msg` parameter as it would any variable, displaying its value to the Output window.

Let's use parameters with our function so that we can set the clip's name, horizontal distance, and vertical distance differently it runs. Example 9-1 shows the revised code.

Example 9-1. A generalized moveClip function

```
function moveClip (theClip, xDist, yDist) {
  theClip._x += xDist;
  theClip._y += yDist;
}
```

We renamed the function from *moveBall( )* to the generic name *moveClip( )* to reflect its ability to work with any clip. We al three parameters: `theClip` (the movie clip we want to move), `xDist` (the horizontal distance to move), and `yDist` (the distance to move). Inside the function body, we use those parameters instead of the literal "hardcoded" values in the earlier *m* example, thereby allowing our function to reposition any clip by any horizontal and vertical distance. Notice that we no longe global variable `mainTimeline` because the function is told which clip to move when it is invoked. We'll cover invoking *n* and specifying its parameter values in the next section.

9.3.2 Invoking Functions with Parameters

When we create a function, we define the parameter names, which are essentially placeholders for data values. When we invo function, we'll provide values to be plugged in for each of the parameters.

The terms *parameters* and *arguments* are used interchangeably in common discussion and in most documentation. Technically are the values used when invoking a function, and *parameters* are the placeholders in the function that "receive" the argument

Recall the generic syntax of function invocation with no arguments:

```
funcName()
```

To supply (or *pass*) arguments to a function, we provide a list of values within the parentheses when invoking the function:

```
funcName(arg1, arg2, arg3,...argn)
```

The values used as arguments can be any legitimate expression in ActionScript, including compound expressions. For examp we defined a simple function, *say( )* , that expects a single parameter, `msg` :

```
function say (msg) {
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

```
  trace("The message is " + msg);
}
```

To invoke *say( )* , we use a statement like this:

```
say("This is my first argument...how touching");
```

or like this:

```
say(99);
```

or even like this:

```
now = new Date( );
say(now)
```

Notice that the values, "This is my first argument . . . how touching", 99, and today's date, belong to different datatypes. Actic allows us to pass any data of any type to a function, as long as the function knows what to do with the passed value. (Languag C and Java require that we predefine the datatype for each parameter, and they display an error if data of the wrong type is pr

Before each argument is passed, its value is fully resolved. Therefore, we can also invoke functions with complex expressions common to use variables or expressions containing variables as arguments to functions. For example:

```
var name = "Gula";
say("Welcome to my web site " + name);
```

Because the expression *"Welcome to my web site " + name* is evaluated before it is passed to the *say( )* function, the function the resolved value "Welcome to my web site Gula". This means that we can programmatically generate function arguments— arguments specified when invoking the function can differ based on things that change at runtime, such as the user's name or This is powerful stuff.

To pass more than one argument to a function, we separate our arguments with commas. Recall that our *moveClip( )* function Example 9-1 accepts three parameters: `theClip` , `xDist` , and `yDist` . Invoking *moveClip( )* , therefore, looks like this:

```
moveClip(ball, 5, -15);
```

Each of the three arguments is assigned as the value of the corresponding parameter named in the function declaration: `ball` to `theClip` , 5 is assigned to `xDist` , and -15 is assigned to `yDist` . The correspondence between parameters and argun based purely on their order in the comma-separated lists; the first argument specified in the function invocation corresponds t parameter in the function declaration, the second argument corresponds to the second parameter, etc. The contents of the argu in the function invocation are passed into the function body, where they are referenced by the parameter names used in the fu declaration. That is, even though the `ball` identifier was used as the argument when invoking the *moveClip( )* function, the movie clip instance is known by the parameter name `theClip` within the function body.

> The arguments used when invoking a function can be any valid expression, including complex expressions literals, or identifiers. The parameters specified in the function declaration and used within the function bo are always identifiers (i.e., the names of variables that are local to the function).

We can move any clip any distance by invoking our generic *moveClip( )* function with different arguments. For example, to r instance named `square` to the right 2 pixels and down 100 pixels, we use:

```
moveClip(square, 2, 100);
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Note that if we want to move the clip horizontally but not vertically, we can simply specify a `yDist` of 0, such as:

```
moveClip(square, 2, 0);
```

Conversely, we can move the clip vertically but not horizontally by specifying an `xDist` of 0:

```
moveClip(square, 0, 100);
```

Therefore, instead of three separate functions, we can use a single function to move the clip horizontally, vertically, or both, b specifying varying arguments in the function invocation. Avoiding unnecessary proliferation of functions keeps our code com maintainable.

To pass a value for a single parameter only, even when multiple parameters are expected by a function, use `undefined` as placeholder for empty parameter values. For example,

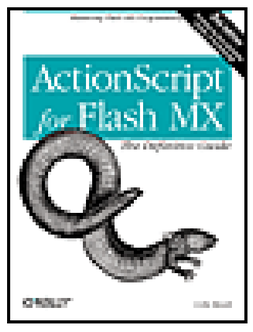
```
function showName (first, middle, last) {
  trace("The name is " + first + " " + middle + " " + last);
}
showName(undefined, undefined, "moock");
```

Of course, you should use this approach only when the function allows empty arguments to be passed. When building a funct must receive a value for a particular parameter, be sure to test for that value inside the function. For example, here we revise t *showName( )* function so that it demands a value for each of its parameters:

```
function showName (first, middle, last) {
  // If the parameters are not valid...
  if (first =  =  = undefined || middle =  =  = undefined || last =  =  = unde
    // Show an error, then quit.
    trace("INVALID ARGUMENT VALUES SPECIFIED");
    return;
  }
  trace("The name is " + first + " " + middle + " " + last);
}
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir



ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Chapter 9. Functions

9.4 Exiting and Returning Values from Functions

Unless instructed otherwise, a function will end naturally when the interpreter finishes executing the last statement in the function's body. You can, however, terminate a function before the last statement is reached. Additionally, a function can return a result (send back a value) to the code that invoked it. Let's see how these things work.

9.4.1 Terminating a Function

The *return* statement, which we introduced in Chapter 6, can be used to terminate a function and, optionally, to return a result. When the interpreter encounters a *return* statement during a function execution, it skips any remaining statements in the function. Consider this example:

```
function say (msg) {
  return;
  trace(msg);          // This line is never reached
}
```

This example is not realistic, because its *return* statement always causes the function to end before the *trace( )* statement is reached. Therefore, the *return* statement is normally the last statement in a function body, unless it is used inside a conditional statement. In the following example, we use *return* to exit the function if the password is not correct:

```
var correctPass = "cactus";
function enterSite(pass) {
  if (pass != correctPass) {
    // Exit if the password is wrong
    return;
  }
  // This code is reached only if the password is correct
  gotoAndPlay("intro");
}
enterSite("hackAttack");  // Function will exit prematurely
enterSite("cactus");      // Function will end naturally
```

As its name implies, *return* tells the interpreter to return control to the script containing the function invocation. If no *return* statement is present, ActionScript acts as if the last line of the function body contains a *return* statement:

```
function say (msg) {
  trace(msg);
  return;              // The return statement is optional in this context
}
```

Regardless of whether the *return* statement is implied or explicit, whenever a function terminates, execution resumes

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

at the line of code following the function invocation. For example:

```
say("Something");  // This executes the code in the say( ) function
// Execution resumes here after the say( ) function terminates
trace("Something else");
```

9.4.2 Returning Values from Functions

As we've seen, *return* always terminates a function. But it can also be used to send a value back to the script that invoked the function, using the following syntax:

```
return expression;
```

The value of *expression* becomes the result of the function invocation. For example:

```
// Define a function that adds two numbers
function combine (a, b) {
  return a + b;  // Return the sum of the two arguments
}
// Invoke the function
var total = combine(2, 1);  // Sets total to 3
```

The expression or result returned by the *return* statement is called the *return value* of the function.

Notice that our *combine( )* function merely calculates and returns the sum of two numbers (it will also concatenate two strings). It does not perform an overt action, as did the *sayHi( )* function (which displayed a message) or the *moveClip( )* function (which repositioned a movie clip).

We can make use of a function's return value by assigning it to a variable:

```
var total = combine(5, 6);                 // Sets total to 11
var greet = combine("Hello ", "Cheryl");  // Sets greet to "Hello Cheryl"
```

The result of a function call is just an ordinary expression. Therefore, it can also be used in additional expressions. This example sets `phrase` to "11 people were at the party":

```
var phrase = combine(5, 6) + " people were at the party";
```

Frequently, we'll use function return values as parts of compound expressions—even as arguments to another function invocation. For example:

```
var a = 3;
var b = 4;
function sqr (x) {  // Squares a number
  return x * x;
}
var hypotenuse = Math.sqrt(sqr(a) + sqr(b));
```

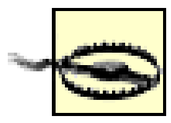
Notice how the example passes the return values of our *sqr( )* function to the *Math.sqrt( )* function! Along the same lines, our earlier example could be rewritten as:

```
var phrase = combine (combine(5,6), " people were at the party");
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

In this example, the inner expression *combine(5,6)*, which evaluates to 11, becomes an argument to the outer *combine( )* function call, where it is concatenated with the string " people were at the party".



If a *return* statement doesn't include an expression to be returned, or if the *return* statement is omitted entirely, a function will return the value `undefined`. In fact, this is a common source of error.

For example, the following won't do anything meaningful, because the *return* statement is missing:

```
function combine (a, b) {
  var result = a + b;  // The result is calculated, but not returned
}
```

Likewise, this too is incorrect:

```
function combine (a, b) {
  var result = a + b;
  return;  // You've forgotten to specify the return value
}
```

The correct function should read:

```
function combine (a, b) {
  var result = a + b;
  return result;
}
```

When creating a function that is supposed to return the result of a calculation, don't forget to include a *return* statement that actually returns the desired value. Otherwise, the return value will be `undefined` and any subsequent calculations based on that result will almost certainly be incorrect.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Chapter 9. Functions

9.5 Function Literals

ActionScript allows us to create *function literals*, which are convenient when we need a function temporarily or when we want to use a function where an expression is expected.

Function literals have the same syntax as standard function declarations, except that the function name is omitted and there's a semicolon after the statement block. The general form is:

```
identifier = function (param1, param2, ...paramn) { statements };
```

where `param1`, `param2`, `...paramn` is an optional list of parameters and *statements* is one or more statements that constitute the function body. Because it doesn't include a function name, a function literal's definition is "lost" unless we store it in a variable (or an array element or object property). We can store a function literal in a variable for later access, like this:

```
// Store a function literal in a variable
var mouseCoords = function () { return [ _xmouse, _ymouse ]; };
// Now we can execute the function
mouseCoords();
```

Note that because ActionScript does not support JavaScript's *Function( )* constructor, dynamic functions cannot be composed at runtime, as they can in JavaScript.

As we'll see in Chapter 12, function literals are often used when assigning methods to classes or objects. For example:

```
Ball.prototype.startMoving = function ( ) {
  // Movement code goes here
};
```

is the same as:

```
Ball.prototype.startMoving = startMoving;
function startMoving ( ) {
  // Movement code goes here
}
```

However, there are three subtle differences between function literals and standard function declarations:

- When assigned as a method, a function literal does not leave an extraneous function reference defined in the scope of the function declaration. For example, in the first *Ball* example (assuming the *Ball* code is attached to the main timeline), a function literal is stored in `Ball.prototype.startMoving` and no separate function reference remains defined on `_root`. But in the second example, a reference to *startMoving( )* exists both in `Ball.prototype.startMoving` and on `_root`.

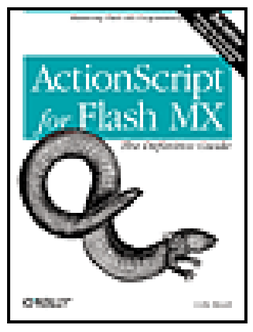
downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

- A standard *function* declaration does not end in a semicolon, whereas a function literal does.
- Functions created with *function* statements are available throughout an entire script, even to statements that come before the function declaration statement (on the same keyframe or within the same *onClipEvent( )* or *on( )* event handler). Function literals are available only to statements that come after the literal in the script.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir



ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Chapter 9. Functions

9.6 Function Availability and Life Span

Like variables, program functions do not live forever and are not directly accessible from everywhere in a movie. In order to invoke functions with confidence, we need to know how to access them from different areas of our movie, and we need to be sure that they exist before we call them. In ActionScript, calling a nonexistent function has no effect. The call fails silently and does not produce an error (in JavaScript, calling a nonexistent function generates an error). To test whether a function ran when you called it, add a trace( ) statement, such as the following, to the function:

```
trace("Function someName executed.");
```

9.6.1 Function Availability

Custom program functions (the functions we create in our code) are *directly* accessible for invocation only from:

- Code attached to the timeline of the movie clip that bears the function declaration
- Code in a Flash 5-style *onClipEvent( )* handler on the movie clip that bears the function declaration
- Code in a Flash 5-style *on( )* event handler on a button on the timeline of the movie clip that bears the function declaration

By "directly accessible," we mean that the function can be invoked simply by name, without reference to a movie clip or object, like this:

```
myFunction();
```

Functions are also *indirectly* accessible (i.e., remotely accessible), using dot syntax, from any point in a movie. As when referring to variables remotely, we must include the path to the movie clip that contains the function, such as:

```
myClip.myOtherClip.myFunction();
_parent.myFunction();
_root.myFunction();
```

So, suppose a clip instance named `rectangle` on the main timeline contains a function named *area( )*. We can invoke *area( )* from anywhere in our movie, using an absolute path to `rectangle`, like this:

```
_root.rectangle.area();
```

To reference a function from a remote movie clip timeline, we follow the same rules used when referring to remote variables, as described in Chapter 2. (We'll also cover more about invoking functions from other movie clips in Chapter 13.)

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Not all functions are attached to movie clip timelines. Some program functions may be attached to user-defined or built-in objects. When attached to an object, a function is accessible only through its host object. We'll discuss the availability and life span of *methods* (i.e., functions attached to objects) in Chapter 12.

To create a function that is directly available throughout an entire movie, define it as a function literal on the `_global` object, as follows:

```
_global.functionName = function ( ) {
  statements
}
```

Global functions and variables were not supported prior to Flash Player 6. For more information on globals, see `_global` in the *Language Reference.*

How do you know where to define your functions? If the function is going to be used just for a single movie clip, define it on the movie clip of interest. If the function is going to be used by multiple movie clips, define it on the main timeline as a global function. We'll discuss code organization when we study object-oriented programming in Chapter 12.

9.6.2 Function Life Span

A function defined on a movie clip timeline is lost when that clip is removed from the Stage. An attempt to invoke a function that was defined in a clip that no longer exists will fail silently. In Flash 5, defining a function on the main movie timeline is the best way to ensure the function's permanence, because the main timeline persists throughout the movie. As of Flash 6, the most permanent location for a function is on the `_global` object, as described in the previous section.

Note that functions in a frame's script are initialized and become available when the playhead first enters the frame containing the script. Therefore, we can use a function invocation before the declaration of the function, as long as both are in the same keyframe's script. For example:

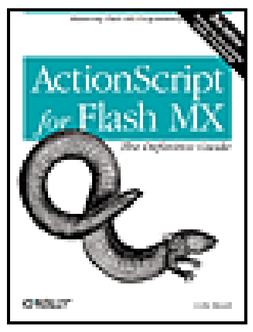
```
// Invoke the tellTime( ) function before its declaration statement
tellTime();
// Declare the tellTime( ) function
function tellTime () {
  var now = new Date();
  trace(now);
}
```

This technique works within a single keyframe's script but not across timeline layers. Code on different layers is executed independently, according to the timeline's order of execution, discussed in Chapter 13.

Because function literals are not available prior to declaration, even in the script in which they are defined, we must define function literals prior to their invocation. As a best practice, you should define functions at the beginning of your scripts. This ensures they will be available for the entirety of the script, even if they are declared as function literals.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir



ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Chapter 9. Functions

9.7 Function Scope

As shown under Section 2.5 in Chapter 2, all ActionScript variables have a certain *scope*, or region of a program within which they are valid. A statement's location in a movie determines:

- Where the interpreter looks to resolve variables referenced by the statement
- The scope of the variables created by the statement

For example, here we issue a statement that creates a variable, `score`:

```
var score = 10;
```

If this statement were attached to `clipA`, the interpreter would set the value of `score` in `clipA`. If the statement were attached to `clipB`, the interpreter would set the value of `score` in `clipB`.

Statements in the body of a function operate in their own, separate scope, called a *local scope*. A function's local scope is private to the function, distinct from the scope of the clip or object to which the function is attached. A new local scope for the function is created each time the function is invoked and destroyed when the function finishes executing. When resolving variables referenced in the statements of the function body, the interpreter looks first in the function's local scope.

Function parameters, for example, are defined in the *local* scope of a function—not on the timeline that bears the function. Hence, parameters are accessible only to the statements of a function's body and only while the function is running. Statements outside the function have no access to the function's parameters.

A function's local scope provides a place for temporary "local" variables used solely within a function. This eliminates potential name conflicts between local function variables and timeline variables, and it reduces overall memory usage.

9.7.1 Functions and the Scope Chain

Even though functions operate in their own local scopes, normal timeline variables are still accessible to the statements of a function body. The local scope of a function is the first place the interpreter looks to resolve variable references, but if the variable reference can't be found in the local scope, the search extends to the movie clip in which the function was declared, and then (as of Flash Player 6) to the global object.

For example, suppose we define a variable, `firstName`, on the timeline of a movie clip, `clipA`. We also declare a function, *getName( )*, on `clipA`. Inside *getName( )*, we refer to the variable `firstName` in a *trace( )* statement:

```
firstName = "Christine";
function getName ( ) {
  trace(firstName);
}
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

```
getName( );
```

When we invoke *getName( )*, the interpreter must find the value of `firstName`. There is no parameter or local variable called `firstName` in the local scope of *getName( )*, so the interpreter checks the timeline of `clipA` for the variable `firstName`. There, the interpreter finds `firstName` and displays its value, "Christine".

Our *trace( )* statement is able to refer to a timeline variable from the body of our *getName( )* function because *getName( )* does not, itself, define a parameter or local variable called `firstName`. Now consider what happens when we add a parameter called `firstName` to the *getName( )* function:

```
firstName = "Christine";
function getName (firstName) {
  trace(firstName);
}
getName("Kathy");
```

This time, when we invoke *getName( )*, we pass "Kathy" as the first (and only) argument, and the interpreter assigns the value "Kathy" to the parameter `firstName`. When the interpreter executes the *trace( )* statement, it searches the local scope, where it finds the `firstName` parameter and its value "Kathy". So, the output of the function this time is "Kathy", not "Christine".

Even though the timeline variable `firstName` exists, the function's parameter called `firstName` takes precedence. The parameter is said to *override* or *occlude* the timeline variable.

Our example demonstrates the operation of the *scope chain*—the hierarchy of objects used by the interpreter to resolve references to variables. For functions attached to timelines, the scope chain has only three levels: the local function scope, the scope of the movie clip that bears the function declaration, and the global scope. (Flash 5 does not have a global scope.) But when we nest functions, the scope chain can involve many more levels, as we'll see later.

To access variables and properties that are not local to the function's scope, we must use dot syntax to form an explicit reference (also called a *qualified reference*). For example:

```
function dynamicGoto () {
  // Deliberately go outside the function's local scope
  _root.myClip.gotoAndPlay(_root.myClip.targetFrame);
}
```

Note that a function's scope chain is determined permanently at compile time, relative to the function *declaration* statement, not any function *invocation* statement. That is, the code in a function's body is scoped to the movie clip that bears the function declaration, not the movie clip that bears the statement that invokes the function. Scope of this type is called *lexical* or *static scope* (as opposed to *dynamic scope*, where the scope can change at runtime). We'll return to the topic of lexical scope when we examine nested functions.

Here's an example showing a misguided use of the scope chain:

```
// CODE ON MAIN MOVIE TIMELINE
// This function's scope chain includes the main movie
function rotate (degrees) {
  _rotation += degrees;
}
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

If we attempt to rotate `clipA` using `_root.rotate`, it rotates the entire main movie, not just `clipA`:

```
// CODE ON clipA's TIMELINE
_root.rotate(30);  // Oops! This rotates the entire movie!
```

In this situation, we can fix our problem by passing the clip to be rotated as an argument to the *rotate( )* function:

```
function rotate (theClip, degrees) {
 theClip._rotation += degrees;
}
// From the _root timeline, invoke rotate( ) with a reference to clipA.
rotate(clipA, 30);

// Or, from clipA's timeline, invoke rotate( ) with a reference to clipA.
_root.rotate(_root.clipA, 30);
```

Alternatively, we can define the *rotate( )* function as a method of the *MovieClip* class so that it has access to the `this` keyword, which refers to the movie clip object through which it was invoked:

```
MovieClip.prototype.rotate = function (degrees) {
  this._rotation += degrees;
}

// CODE ON clipA's TIMELINE
this.rotate(30);
```

> Functions that operate on a movie clip are best defined as methods of the *MovieClip* class, as discussed in Chapter 12 under Section 12.8.4.

Due to a bug in Flash Player 5, function literals assigned as movie clip methods were not lexically scoped. For example:

```
// Create a variable on the current timeline.
var x = 10;
// Create a variable on theClip's timeline.
theClip.x = 20;
// Assign a function literal as a method of theClip.
theClip.traceX = function ( ) {
  // Flash Player 5 erroneously displays the value of x in theClip: 20
  // Flash Player 6 correctly displays the value of x on the timeline
  // that bears the function declaration: 10
  trace("x is: " + x);
}
// Invoke the traceX( ) method.
theClip.traceX( );
```

For the sake of backward compatibility, Flash Player 6 applies the correct scoping behavior only with Flash 6 format *.swf* files. The behavior of Flash 5 *.swf* files does not change when they are played in Flash Player 6.

9.7.2 Local Variables

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Variables assigned to a function's local scope are called *local variables*. Local variables, including parameters, are accessible only to statements in the body of the function in which they are defined and exist only while that function runs. To create a local variable (other than the parameters that automatically become local variables), we use the *var* statement inside any function, like this:

```
function funcName () {
  var temp = "just testing!";  // Declares the local variable temp
}
```

Local variables are useful for holding information temporarily. Here, for example, we use the local variable `lastSpacePlusOne` to hold an interim result. Like all local variables, it dies when the function ends:

```
function getLastWord (text) {
  var lastSpacePlusOne = text.lastIndexOf(" ") + 1;                // Local
  var lastWord = text.subString(lastSpacePlusOne, text.length);   // Local
  return lastWord;
}
// Displays: "word"
trace(getLastWord("Tell me the last word"));
// Displays: undefined. lastSpacePlusOne is local and not
// available outside the getLastWord( ) function.
trace(lastSpacePlusOne);
```

When local variables expire at the end of a function, the memory associated with them is marked for automatic deletion. By using local variables to store all temporary values, we can conserve memory in a program. Furthermore, when we declare a local variable, we need not worry that it might conflict with a timeline variable of the same name.

Of course, not all variables used in functions need to be local. Earlier we saw that we can create, read, and set timeline variables from inside a function. Within a function, variable assignment statements that do not apply to a local variable are scoped to the timeline that bears the function declaration. In this example, `x` is a local variable, but `y` and `z` are timeline variables:

```
var z = 1;

function createVars () {
  var x = 10;    // Create local variable x
  y = 13;        // Create timeline variable y
  z = 2;         // Alter timeline variable z
}
createVars();    // Call the function
trace(x);        // x is undefined (x dies when the function ends)
trace(y);        // y is 13 (y exists after the function ends)
trace(z);        // z is 2 (z was permanently altered by the function)
```

The rules for creating timeline variables apply even if our function is a method of an object. We won't cover objects until Chapter 12, but for those familiar with object-oriented programming, Example 9-2 shows code that proves the point. Observe that `x` is defined on the timeline because it doesn't exist in the local scope of *newFunc( )*.

Example 9-2. Variable scope within object methods

```
newObj = new Object();                          // Create an object
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

```
newObj.newFunc = function () { x = 12; };  // Attach a new method
newObj.newFunc();                          // Call the method

// Now let's check for x
trace("x is " + x);                        // x is 12
trace("newObj.x is " + newObj.x);          // newObj.x is undefined
```

Using unqualified variable references inside functions to affect timeline variables may appear handy, but more often than not it is confusing, particularly in large projects. It's wise to take the ambiguity out of variable access by using qualified variable references (i.e., prefix the variable with an explicit object reference, such as `this.x` versus `x`, as discussed in Chapter 2 under Section 2.5.4).

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Chapter 9. Functions

9.8 Function Parameters Revisited

Now that we're comfortable with how functions work, let's return to the topic of function parameters. Our current discussion requires a little knowledge of objects, so new programmers may want to read Chapter 12 before reading this section.

9.8.1 Number of Parameters

Earlier we saw that the parameters of a function are declared when the function is created. Recall the syntax:

```
function funcName (param1, param2, ...paramn) {
  statements
}
```

Perhaps surprisingly, the number of parameters passed to a function can differ from the number specified in the formal function declaration. Functions can accept any number of parameters, whether more than or fewer than the "expected" number. When a function is called with fewer than the declared number of parameters, the value of each missing parameter is set to `undefined`. For example:

```
function viewVars (x, y, z) {
  trace ("x is " + x);
  trace ("y is " + y);
  trace ("z is " + z);
}
viewVars(10);  // In Flash Player 6, displays:
               // x is 10
               // y is undefined
               // z is undefined
               // (In Flash Player 5, undefined variables are displayed as a
               // blank string in the Output window.)
```

When a function is called with *more* parameters than the declared number, excess parameter values can be accessed using the *arguments* object. (Obviously, the excess parameters can't be accessed by name, as explicitly declared parameters can, because their names were not declared.)

9.8.2 The arguments Object

During the execution of any function, the built-in *arguments* object gives us access to the following:

- The number of parameters that were passed to the function
- An array containing each of those parameter values

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

- A reference to the function being executed
- A reference to the calling function, if any

The *arguments* object is really a special hybrid between an array and an object with some other properties. In Flash 5, *arguments* was not an array instance. In Flash MX, however, *arguments* is an instance of the *Array* class and therefore supports *Array*'s methods and properties.

9.8.2.1 Retrieving parameter values from the arguments array

The `arguments` array lets us check the value of any function invocation argument, whether or not that parameter is defined formally in the function's declaration statement. To access a parameter, we examine the indexes of the `arguments` array:

```
arguments[n]
```

where *n* is the index of the argument we're accessing. The first argument (the leftmost argument in the function-call invocation) is stored at index 0 and is referred to as `arguments[0]`. Subsequent arguments are stored in order, proceeding to the right—so, the second argument is `arguments[1]`, the third is `arguments[2]`, and so on.

From within a function, we can tell how many arguments were passed to the currently executing function by checking the number of elements in `arguments`, as follows:

```
numArgs = arguments.length;
```

We can easily cycle through all the arguments passed to a function and display the results in the Output window, as shown in Example 9-3.

Example 9-3. Displaying an unknown number of arguments

```
function showArgs () {
  for (var i = 0; i < arguments.length; i++) {
    trace("Argument " + (i + 1) + " is " + arguments[i]);
  }
}
showArgs(123, 23, "skip intro");
// Displays...
Argument 1 is 123
Argument 2 is 23
Argument 3 is skip intro
```

The `arguments` array allows us to create very flexible functions that accept an arbitrary number of arguments.

Here's a generic function that removes any number of duplicated movie clip instances from the Stage:

```
function killClip () {
  for (var i = 0; i < arguments.length; i++) {
    arguments[i].removeMovieClip();
  }
}
killClip(clip10, clip5, clip13);
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Reader Exercise: Modify our earlier *combine( )* function to accept an arbitrary number of inputs. What other functions might benefit from accepting an arbitrary number of parameters? How about a function that averages a list of numbers? (Hint: sum all the arguments and then divide by the number of arguments.)

9.8.2.2 The callee and caller properties

As we've seen, the `arguments` array lets us retrieve a function's arguments. The *arguments* object has two properties: `callee`, which stores a reference to the executing function, and `caller`, which stores a reference to the function that called the current function (or `null` if no function called the current function).

Normally, we know the name of the function we're calling, but if we are executing an anonymous function that was originally created with a function literal, the `callee` property can prove useful. Example 9-4 shows a function, created with a function literal, that performs recursive executions without knowing its own name.

Example 9-4. Counting down with callee

```
count = function (x) {
  trace(x);
  if (x > 1) {
    arguments.callee(x - 1);
  }
}
count(25);
```

Obviously we can count down without using recursive anonymous functions. We'll see a more realistic example of function recursion in Section 9.9 later in this chapter. Using `arguments.callee` to refer to the current function (rather than the function's actual name) alleviates the need to update our code if the function's name changes.

We can use `arguments.caller` to execute or identify a calling function, as described in the *Language Reference.*

9.8.3 Passing Parameters by Value Versus by Reference

There's one more parameter subtlety we should consider—the difference between passing data to a function *by value* and *by reference*.

When we pass a primitive data value as an argument to a function, the function receives a copy of the data, not the original. Changes made to a parameter in the function have no effect on the original argument outside that function. In Example 9-5, `variableName`'s value is initially set to 25. Changing its value to 10 within the *setValue( )* function has no effect on `y`'s value.

Example 9-5. Primitive data is passed by value

```
var y = 25;
function setValue (variableName) {
  variableName = 10;
}
setValue(y);
trace("y is " + y);  // Displays: "y is 25"
```

Primitive data is therefore said to be *passed by value.* When we pass *composite* data as an argument to a function, however, the function receives a *reference* that points to the same data as the original argument, not just a duplicate of the

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

data. Altering the data via the parameter variable affects the original data and therefore affects other variables that point to the same data, even outside of the function. Composite data is therefore said to be *passed by reference*.

In Example 9-6, changes to the `myArray` parameter variable affect the external `boys` array because they both point to the same data in memory.

Example 9-6. Modifying composite data arguments passed by reference

```
// Create an array
var boys = ["Andrew", "Graham", "Derek"];

// setValue( ) sets the value of the first element of an array
function setValue (myArray) {
  myArray[0] = "Sid";         // Set the first element of the array
}

// Pass our array to the function
setValue(boys);

// Check the value of our array elements
trace("Boys: " + boys);  // Displays: "Boys: Sid,Graham,Derek"
```

Note that new values assigned to individual elements of the array from inside a function are reflected outside the function as well. However, suppose that instead of assigning new values to one or more elements, we assign a new value to the parameter itself. Doing so will break its association with the original argument. Subsequent changes to the parameter variable will have no effect on the original argument, as shown in Example 9-7. In this example, although the `boys` array is passed as an argument, the `myArray` parameter variable is immediately set to `girls`. Subsequent changes to `myArray` affect the `girls` array, not the `boys` array.

Example 9-7. Breaking the association between an argument and a parameter

```
// Create two arrays
var boys  = ["Andrew", "Graham",  "Derek"];
var girls = ["Alisa",  "Gillian", "Daniella"];

// setValue( ) ignores the passed array and modifies the girls array
function setValue (myArray) {
  myArray = girls;      // Make myArray point to girls, not boys

  myArray[0] = "Mary";  // Changes the first element of girls
}

// Pass the boys array to the setValue( ) function
setValue(boys);
trace("Boys: " + boys);   // Displays: "Boys: Andrew,Graham,Derek"
trace("Girls: " + girls); // Displays: "Girls: Mary,Gillian,Daniella"
```

More information on primitive and composite data can be found in Chapter 3.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Chapter 9. Functions

9.9 Recursive Functions

A *recursive* function is a function that calls itself (by using its own name within its function body). Here's a simple example that shows the principle of recursion. But because the code tells the *trouble( )* function to execute repeatedly (like an image reflected infinitely in two opposing mirrors), Flash will quickly exceed its maximum recursion limits, causing an error:

```
function trouble () {
  trouble();
}
```

A recursive function that calls itself unconditionally, such as *trouble( )* does, is said to exhibit *infinite recursion* (i.e., the function never stops calling itself). Infinite recursion will cause the computer to run out of memory quickly. To prevent this from happening, Flash sets a maximum of 256 levels of recursion, at which point the recursive function call will fail.

Practical recursive functions call themselves only while a given condition is met (thus preventing infinite recursion). Example 9-4 used recursion to count from a specified number down to 1, but obviously that can be accomplished without recursion.

One classic use of recursion is to calculate the mathematical *factorial* of a number. The factorial of 3 (written as 3! in mathematical nomenclature) is 3 * 2 * 1 = 6. The factorial of 5 is 5 * 4 * 3 * 2 * 1 = 120. Example 9-8 shows a factorial function that uses recursion.

Example 9-8. Calculating factorials using recursion

```
function factorial (x) {
    if (x < 0) {
      return undefined;  // Error condition
    } else if (x <= 1) {
      return 1;
    } else {
      return x * factorial(x-1);
    }
}
trace (factorial(3));  // Displays: 6
trace (factorial(5));  // Displays: 120
```

As usual, there is more than one way to skin a proverbial cat. Using a loop, we can also calculate a factorial without recursion, as shown in Example 9-9.

Example 9-9. Calculating factorials without recursion

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

```
function factorial (x) {
    if (x < 0) {
      return undefined; // Error condition
    } else {
      var result = 1;
      for (var i = 1; i <= x; i++) {
        result = result * i;
      }
      return result;
    }
}
```

Examples Example 9-8 and Example 9-9 represent two ways of solving the same problem. The recursive method says, "The factorial of 6 is 6 multiplied by the factorial of 5. The factorial of 5 is 5 multiplied by the factorial of 4 . . ." and so on. The nonrecursive method loops over the numbers from 1 to `x` and multiplies them all together into one big number.

Which approach is better—recursive or nonrecursive—depends on the problem. Some problems are solved more easily using recursion, but recursion can be slower than nonrecursive solutions. Recursion is best used when you don't know how deeply a data structure may be nested. For example, suppose you wanted to list all the files within a subdirectory, including listing all files within any nested subdirectory, ad infinitum. It would be inconvenient to write a general solution that worked for any number of subdirectories without resorting to recursion. A recursive solution might look like this in pseudocode:

```
function listFiles (directoryName) {
  do (check the next item in directoryName) {
    if (this item is a subDirectory itself) {
      // Recursively call this function with the new subdirectory
      listFiles(subDirectoryName);
    } else {
      // Display the name of this file
      trace(filename);
    }
  } while(there are still items to check);
}
```

When we consider the *XML* object the ActionScript Language Reference, we'll use recursion to list all the elements in an XML document.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Chapter 9. Functions

9.10 Nested Functions

In ActionScript, functions can be declared within functions. Example 9-10 creates a function, *showFullName( )*, with two nested functions, *showFirstName( )* and *showLastName( )*.

Example 9-10. A function with two nested functions

```
function showFullName ( ) {
  function showFirstName ( ) {
    trace("Colin");
  }
  function showLastName ( ) {
    trace("Moock");
  }

  showFirstName( );
  showLastName( );
}

showFullName( );
```

Inside *showFullName( )*, the nested functions can be invoked as usual. However, like a local variable, a nested function is accessible *only* to its parent function. Code outside the parent function cannot execute the nested function.

Nested functions are often used to encapsulate a series of complex operations into a single package, while preventing the outside world from gaining access to its internal workings. For an interesting example of nested functions (sometimes called *inner functions*) used to simulate Java-style private members, see:

http://www.crockford.com/javascript/private.html

9.10.1 Nested Function Scope

When a nested function executes, the interpreter adds the nested function's local scope to the scope chain of its parent function. For example, in Example 9-10 the scope chain of *showFullName( )* is:

- Global object (interpreter ends its search for variables here)
- Enclosing movie clip object (where *showFullName( )* was defined)
- *showFullName( )* local scope (interpreter starts its search for variables here)

And the scope chain of *showFirstName( )* is

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

- Global object (interpreter ends its search for variables here)
- Enclosing movie clip object (where *showFullName( )* was defined)
- *showFullName( )* local scope
- *showFirstName( )* local scope (interpreter starts its search for variables here)

The scope of the nested function is the same as its parent's with the addition of its own local scope. If a variable does not exist in *showFirstName( )*'s local scope, the interpreter will check *showFullName( )*'s local scope, and so on up the chain. Furthermore, all variables defined in an outer function are available to nested functions, even after the outer function finishes executing. In the following example, the function *enableZoom( )* creates an *onMouseDown( )* event handler on `theClip`, using a nested function literal.

```
function enableZoom (theClip) {
  var zoomFactor = 1.5;
  theClip.onMouseDown = function ( ) {
    this._xscale *= zoomFactor;
    this._yscale *= zoomFactor;
  }
}

// Enable zooming for the current movie clip.
enableZoom(this);
```

The *onMouseDown( )* handler will execute whenever the mouse is clicked—long after *enableZoom( )* has finished its own execution. However, even though *enableZoom( )* has finished executing, its `zoomFactor` variable continues to be available to the nested *onMouseDown( )* function via the scope chain.

Here are two more advanced examples showing the power of nested function scope.

Example 9-11 creates a new global function, *setTimeout( )*, that invokes a supplied function, *`func`*, after *`delay`* milliseconds. A nested function, *callFunc( )*, does the work of invoking *`func`* (which is passed as a parameter to *setTimeout( )*). Via the scope chain, *callFunc( )* can refer to both *setTimeout( )*'s *`func`* parameter and its local `id` variable.

Example 9-11. Remembering an interval ID

```
_global.setTimeout = function (func, delay) {
  var id = setInterval(callFunc, delay);
  function callFunc ( ) {
    clearInterval(id);
    func( );
  }
}

// Usage:
setTimeout(sayHi, 1000);

function sayHi ( ) {
  trace("hello world");
}
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Example 9-12 is a variation on the same theme. It uses the scope chain to access the parameters of *slideTo( )* from the nested function assigned to the calling movie clip's *onEnterFrame( )* event handler.

Example 9-12. Implementing a slideTo( ) method

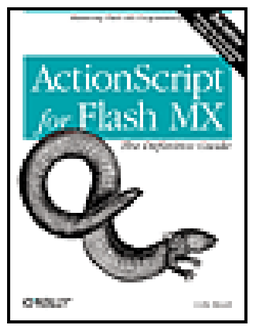
```
// Method: MovieClip.slideTo( )
//   Desc: Animates a clip to the point (leaderX, leaderY) at
//         the specified speed.
// Params: leaderX     The x-coordinate to move to.
//         leaderY     The y-coordinate to move to.
//         speed       Rate of movement (pixels per frame).
MovieClip.prototype.slideTo = function (leaderX, leaderY, speed) {
  // Only move if the clip's not at the destination.
  if (this._x != leaderX && this._y != leaderY) {
    // Create a nested function and assign it as the clip's
    // onEnterFrame( ) event handler.
    this.onEnterFrame = function ( ) {
      // Determine distance between clip and destination.
      // The parameters leaderX and leaderY are accessed via
      // the scope chain.
      var deltaX = this._x - leaderX;
      var deltaY = this._y - leaderY;
      var dist = Math.sqrt((deltaX * deltaX) + (deltaY * deltaY));

      // Allocate speed between x and y axes.
      var moveX = speed * (deltaX / dist);
      var moveY = speed * (deltaY / dist);

      // If the clip has enough speed to overshoot the destination,
      // just go to the destination. Otherwise, move according to
      // the speed specified by the slideTo( ) parameter.
      if (speed >= dist) {
        this._x = leaderX;
        this._y = leaderY;
        // The clip has arrived, so stop remove onEnterFrame( ).
        delete this.onEnterFrame;
      } else {
        this._x -= moveX;
        this._y -= moveY;
      }
    }
  }
}

// Example Usage:
// Slides the clip to the mouse position when the mouse is clicked.
theClip_mc.onMouseDown = function ( ) {
  this.slideTo(_root._xmouse, _root._ymouse, 10);
}
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir



Chapter 9. Functions

9.11 Built-in Functions

Although we've seen how to create user-defined functions, let's not forget that ActionScript comes with a bevy of built-in functions (akin to verbs in our language analogy). We've already seen some built-in functions that allow us to manipulate data. We've also touched on functions that control the Flash movie and the user environment.

For example, to manipulate the playhead of a movie clip, we can call the *gotoAndPlay( )* function with a frame number as a parameter:

```
gotoAndPlay(5);
```

If you are a new programmer, you may be experiencing an epiphany. I hope you have noticed that you invoke built-in functions using the function-call operator (the parentheses) and an argument list (the value 5 in this case) just like our custom user-defined functions! Built-in functions, such as *gotoAndPlay( )*, are used just like the functions we've been declaring ourselves. Custom functions typically do something that a built-in ActionScript function doesn't already offer. Like any custom function, each built-in function has a name, zero or more optional or required arguments, and a return value (although sometimes the value is `undefined`).

Even though Flash has long referred to *gotoAndPlay* as an "Action," we now see it in its true form, as a built-in function. In Chapter 6, you learned that some Flash Actions are statements and some are functions. Now that you've studied both thoroughly, you'll be able to tell which are which.

By understanding certain Actions as *functions*, you'll have an easier time remembering and using their syntax. For example, to load a movie into the Flash Player we need to know that a *loadMovie( )* function exists and what parameters it expects. After a quick jaunt over to the *Language Reference*, we find the info and can easily put together a statement like this:

```
loadMovie("myMovie.swf", "_level1");
```

This will be easy stuff once you've become comfortable with using functions.

ActionScript's built-in functions are many and varied. They give us control over the elements of a movie, empowering us to examine and change everything from the volume of a sound to the amount of text selected in an editable text field. For a thorough description of ActionScript's built-in functions, consult the ActionScript Language Reference. Skim it periodically to familiarize yourself with the type of functions that are available, although there is no need to memorize their specific syntax. In addition to the built-in functions that are a native part of the ActionScript language, Macromedia and third parties offer libraries of custom functions that can be incorporated into your movies and used exactly like built-in functions. For more information on extending Flash, see the resources in Appendix A.

9.11.1 Built-in Function Availability

Some built-in functions are universally available, while others can be invoked only through an object. If the function is a *global* function, it can be used anywhere; if the function is a *method* of an object, however, it must be used in conjunction with the appropriate object. Because we haven't covered objects yet and most of ActionScript's built-in

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

functions are object methods, we'll delay our consideration of the built-in methods until Chapter 12.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Chapter 9. Functions

9.12 Functions as Objects

In ActionScript, functions are technically a special type of built-in object. The *Function* class, formally exposed to ActionScript in Flash Player 6 and documented in the ActionScript Language Reference, creates individual function objects. Let's see what that means and how it affects what you can do with functions.

9.12.1 Passing Functions to Functions

We can use any function as an argument to another function, like this:

```
function1(function2);
```

Note that if there are no parentheses following *function2*, the interpreter doesn't execute *function2* but instead just passes its "object reference" as an argument to *function1*. That is, *function1* receives a reference to *function2* itself, not the return value of *function2( )*. Because objects are passed by reference, we can pass a function identifier to another function and it will arrive unscathed. The passed function can be executed like this:

```
function doCommand (command) {
  command( );      // Executes the passed function
}

// Some examples:
doCommand(stop);  // Pass the built-in stop( ) function (stops current movie)
doCommand(play);  // Pass the built-in play( ) function (plays current movie)
```

We can use this technique to assign a callback function to a series of text field objects:

```
function initFormFields (fields, callbackFunc) {
  // Set focus callback functions for text fields.
  for (var i = 0; i < fields.length; i++) {
    fields[i].onSetFocus = callbackFunc;
  }
}

// Create a list of the text fields.
editFields = [firstNameField, lastNameField, emailField,
                        phoneField, companyField, commentsField];

// Function that colors a text field blue.
function handleFieldFocus ( ) {
  this.background = true;
  this.backgroundColor = 0x333399;
}
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

```
initFormFields(editFields, highlightField);
```

9.12.2 Storing Functions in Variables

Because functions are a type of object, we can treat them as we treat any other data. In the following example, we assign the built-in *gotoAndPlay* function to the variable `gp`, which gives us a shorter way to refer to the function:

```
var gp = gotoAndPlay;  // Create a shortcut reference to gotoAndPlay
gp(25);                // Invoke gotoAndPlay( ) using our reference
```

Note again the omission of the parentheses after *gotoAndPlay* in the first line of the example. As we'll see in Chapter 10 and Chapter 12, functions are commonly stored in object properties to create event handlers and methods.

9.12.3 Attaching Properties and Methods to Functions

In addition to passing and storing functions as objects, we can exploit the "objectness" of functions by attaching properties to them, like this:

```
// Create a function
function myFunction () {
  trace(myFunction.x);
}

// Attach a property to it
myFunction.x = 15;

// Check the property value by invoking the function
myFunction();  // Displays: 15
```

> By attaching properties to functions, we can maintain the state of a piece of information between function executions without cluttering up a timeline with variables. However, properties should normally be stored on objects rather than functions.

Function properties offer the benefits of local variables without expiring between function invocations. This is useful when a function needs to be invoked with a unique identifier. Using function properties allows the function to maintain state. Although a function typically "forgets" everything that happened in previous invocations, function properties allow the function to "remember" what happened previously. This is most useful when you want to keep track of how many times an operation has been performed. Here, for example, is a generic function that duplicates a movie clip and gives the duplicated clip a unique name and level by remembering how many clips we've already assigned in the `makeClip.count` variable:

```
makeClip.count = 0;    // Define a property of makeClip( ) (remember that
                       // makeClip( ) already exists because functions are
                       // defined before code runs)

// Duplicate a passed clip and assign the new clip an automatic name
function makeClip (theClip) {
  // Add one to our clip counter
  makeClip.count++
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

```
  // The previous line can be written more portably as:
  // arguments.callee.count++

  // Now duplicate the clip, assigning it a unique name and depth
  theClip.duplicateMovieClip(theClip._name + makeClip.count, makeClip.count);
}

makeClip(square);   // Make a duplicate of square using makeClip( )

square1._x += 100;  // Now move the duplicated square to the right
```

A variable that maintains its value between function invocations is called a *static variable* in some languages, such as C. Static variables should not to be confused with so-called *static functions* in C, in which the word static denotes "private." Note that we initialize the `makeClip.count` variable to 0 once at the start of our script, but it is not reset (reinitialized) every time the function is called. As a general rule, function properties should be used only to store information that pertains to the function. More general information should be stored either in a class or as a global variable.

ActionScript functions can also have other functions attached to them, which can be used to simulate Java-style static class methods. For example:

```
function SomeClass ( ) {
  // Constructor code goes here.
}
SomeClass.staticMethod = function ( ) {
  // Static method code goes here.
}
```

where `SomeClass` is the name of the class and `staticMethod` is the name of the static method.

For examples of function methods, see the built-in *Function.call( )* and *Function.apply( )* methods in the ActionScript Language Reference. These methods execute a function in the context of a specified object. We'll revisit static methods and the `prototype` property in Chapter 12.

9.12.4 Returning Functions from Functions

Finally, because function objects are data, they can be returned by functions. The following example shows a function, *makeGotoHandler( )*, that returns a nested function literal for use as the callback of a movie clip event handler. The function literal sends the clip's playhead to the frame specified by `frameNumber`.

```
function makeGotoHandler (frameNumber) {
  return function ( ) {
    this.gotoAndStop(frameNumber);
  };
}

// Usage:
_root.onMouseDown = makeGotoHandler(30);
_root.onMouseUp = makeGotoHandler(40);
```

Using *makeGotoHandler( )*, we can conveniently create a callback function without the work of actually typing out the

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

function each time we want to assign it. Notice that we're once again using the scope chain to retrieve the value of `frameNumber` from the event handler callback, long after *makeGotoHandler( )* has finished executing. For details, see Section 9.10.1, earlier in this chapter.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Chapter 9. Functions

9.13 Centralizing Code

Probably the single most important feature of functions is their reusability—we can create reusable segments of code that can be executed from anywhere in a movie. The mark of good code is that it is modularized and centralized. Modules can be replaced without affecting other modules or calls to the updated module. Furthermore, the code is in one central place that makes it easy to find and maintain. Importantly, there aren't multiple scripts in which identical or nearly identical code is repeated. Instead, function calls are used to perform operations that are centralized in a library of generalized functions. Code modularity and centralization make it easier to reuse all or part of the code in other projects. Because functions can be executed from other movie clips, they can be stored in a central location, easing code maintenance and upgrades.

Suppose, for example, that we have three buttons in three different movie clips, all with the same purpose. Instead of triplicating our code by adding it to each button, we can put the code in a function on the main timeline and invoke it from each button as needed. This saves time, reduces the potential for errors, and makes it easy to update the behavior of all three buttons at once. Because the code is centralized, testing is more reliable, and troubleshooting is much easier. If one button works and a second doesn't, chances are that the problem is in the second button's function invocation and not in the centralized behavior.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Chapter 9. Functions

9.14 The Multiple-Choice Quiz Revisited

In Example 1-1, we introduced new programmers to a simple scripted movie—a multiple-choice quiz. Let's revisit that quiz now to see how we can centralize its code.

Our new quiz's layer structure is set up exactly as before. We'll be changing the code on only the first frame, the `quizEnd` frame, and the buttons.

You can obtain the *.fla* file for both the original and revised versions of the quiz from the online Code Depot.

9.14.1 Organizing the Quiz Code into Functions

In our first attempt at creating a multiple-choice quiz, we scattered our code around our movie. We placed the logic of our quiz in three places: on the first frame, where we initialized our quiz; on the `q2` frame, where we assigned Question 2's button event handlers; and on the `quizEnd` frame, where we tallied the user's score. We're now going to centralize the quiz logic by performing all of those tasks in the first frame. To initialize the quiz, we'll still use a simple series of statements as we did in the first version, but to track the user's answers and tally the user's score, we'll use two functions, *answer( )* and *gradeUser( )* . We'll also use function literals to define the behavior of the quiz answer buttons.

Example 9-13 shows the code on our quiz's first frame. This is where we'll store all the logic needed to run our quiz. Take a look at the code in its entirety—then we'll dissect it. This code does not yet reflect best practices, such as using arrays to store a series of related data items. Much of the code is still only intended for illustration purposes—in this case, to learn how to centralize code using functions. We'll refine the example further later in the book.

Example 9-13. A multiple-choice quiz, version 2

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

//  Init quiz variables
var quizTimeline = this;     // Store a reference to the timeline that
                             // contains the quiz user interface
var numQuestions = 2;        // Number of questions in the quiz
var q1answer;                // User's answer for question 1
var q2answer;                // User's answer for question 2
var correctAnswer1 = 3;      // The correct choice for question 1
var correctAnswer2 = 2;      // The correct choice for question 2
answer.currentQuestion = 0;  // Function property: the question being answered

//   Function: answer( )
//       Desc: Registers the user's answers to quiz questions and
//             advances the quiz through its questions
//     Params: choice           The user's answer for the current question
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

```
// Properties: currentQuestion  The question being answered
function answer (choice) {
  // The answer( ) function is called once per question. We add one to
  // currentQuestion each time answer( ) is called so we can track the
  // question number we're on. Because currentQuestion is a function property,
  // it maintains its value between function calls.
  answer.currentQuestion++;

  // Display the question and answer in the Output window for debugging.
  trace("Question " + answer.currentQuestion
        + ". The user answered: " + choice + ".");

  // Record the user's answer to this question. The set( )
  // function dynamically constructs a variable name in the
  // series q1answer, q2answer,...qnanswer.
  set("q" + answer.currentQuestion + "answer", choice);

  // If we're on the last question...
  if (answer.currentQuestion =  = numQuestions) {
    // ...go to the quizEnd frame.
    quizTimeline.gotoAndStop("quizEnd");
    // And grade the user.
    gradeUser( );
  } else {
    // ...otherwise, go to the next question frame.
    quizTimeline.gotoAndStop("q"+ (answer.currentQuestion + 1));
  }
}

//   Function: gradeUser( )
//       Desc: Tallies the user's score at the end of the quiz.
function gradeUser ( ) {
  // Report that we're about to grade the quiz in the Output window.
  trace("Quiz complete. Now grading...");

  // Create a local variable to track the
  // number of questions user answered correctly.
  var totalCorrect = 0;

  // Count how many questions the user answered correctly.
  // For each question...
  for (var i = 1; i <= numQuestions; i++) {
    // If the user's answer matches the correct answer.
    if (eval("q" + i + "answer") =  = eval("correctAnswer" + i)) {
      // Give the user a point.
      totalCorrect++;
    }
    // Display the correct answer and the user's answer
    // in the Output window for debugging.
    trace("Question " + i
          + ". Correct answer: " + eval("correctAnswer" + i)
          + ". User's answer: "  + eval("q" + i + "answer"));
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

```
}

// Display the final score in the Output window for debugging.
trace("User's score: " + totalCorrect + "/" + numQuestions);

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

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

// Customize the font face, size, and color for the text field.
var formatObj = new TextFormat( );
formatObj.size = 16;
formatObj.color = 0xC7FF9C;
formatObj.font = "_sans";
formatObj.bold = true;
quizTimeline.totalOutput_txt.setTextFormat(formatObj);
}

// Code executed when button 1 is pressed.
choice1_btn.onRelease = function ( ) {
  // Call answer( ), which records the user's choice (supplied as an argument)
  // and advances the quiz to the next question.
  this._parent.answer(1);
};

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

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

Our first chore is to stop the movie, using the *stop( )* function, so that it doesn't play through all the questions. Next, we initialize our quiz's timeline variables. You'll recognize `q1answer` and `q2answer` from the first version of the quiz. We've also added:

- `numQuestions` , which we'll use in our *answer ( )* and *gradeUser ( )* functions.
- `correctAnswer1` and `correctAnswer2` , which we'll use when grading the quiz.
- `answer.currentQuestion` , a function property that indicates which question is being answered when *answer( )* is called.
- `quizTimeline` , which is a reference to the movie clip in which the quiz interface will be displayed. We use `quizTimeline` rather than `_root` so that our movie can be loaded safely into a movie clip without breaking.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

With our variables initialized, we can create the *answer( )* function, which records the user's answers and advances the playhead to the next question. The *answer( )* function expects one parameter—`choice` , which is the number of the user's answer for each question—so its function declaration begins like this:

```
function answer (choice) {
```

Each time an answer is given, the function increments `currentQuestion` , a function property that tracks the question being answered:

```
answer.currentQuestion++;
```

Next, we set the user's choice in a dynamically named timeline variable that corresponds to the question being answered. We use the value of the `currentQuestion` property to determine the name of our timeline variable—`q1answer` , `q2answer` , etc. (in a later version of the quiz, we'll use a more elegant array):

```
set ("q" + answer.currentQuestion + "answer", choice);
```

With the user's choice stored in the appropriate variable, we need to decide what to show the user next:

- If we are on the last question, we display the `quizTimeline` 's `quizEnd` frame and run the *gradeUser( )* function.
- Otherwise, we display the next question, which is at the frame labeled *"q" + (answer.currentQuestion + 1)* :

```
if (answer.currentQuestion =  = numQuestions) {
  quizTimeline.gotoAndStop("quizEnd");
  gradeUser( );
} else {
  quizTimeline.gotoAndStop("q"+ (answer.currentQuestion + 1));
}
```

That takes care of our question-answering logic. The *answer( )* function is ready to handle answers from any question in the quiz. Now let's build the function that evaluates those answers: *gradeUser( )* .

The *gradeUser( )* function takes no parameters. It has to compare each user answer with each correct answer and display the user's score. To tally the user's score, we use the variable `totalCorrect` , which was a timeline variable in the first version of the quiz but is more appropriately a local variable in this version (`totalCorrect` is used only by the *gradeUser( )* function, so it is best kept private to that function).

We compare the user's answers with the correct answers in a *for* loop—we cycle through the loop body once for each question in the quiz:

```
for (i = 1; i <= numQuestions; i++) {
```

Inside the loop, a comparison expression tests the user's answers against the correct answers. Using *eval( )* , we dynamically retrieve the value of each user-answer variable and each correct-answer variable. (In a later version of the quiz, we'll get rid of the *eval( )* statement by using an array.) If the two variables are equal, `totalCorrect` is incremented:

```
if (eval("q" + i + "answer") =  = eval("correctAnswer" + i)) {
  totalCorrect++;
}
```

When the loop finishes, `totalCorrect` will contain the number of questions that the user answered correctly. We display that number in a dynamically created text field:

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

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

where `totalOutput_txt` is the name of the text field, 1 is the field's depth in `quizTimeline` 's content stack, 150 and 200 are the field's (*x* , *y*) position, and 200 and 20 are its height and width. To display the value of `totalCorrect` in the text field, we assign it to the `text` property:

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

Finally, we apply custom formatting to the text field. See the *TextFormat* class in the *Language Reference* for details.

```
var formatObj = new TextFormat( );
formatObj.size = 16;
formatObj.color = 0xC7FF9C;
formatObj.font = "_sans";
formatObj.bold = true;
quizTimeline.totalOutput_txt.setTextFormat(formatObj);
```

Voila! With both *answer( )* and *gradeUser( )* created and our quiz variables initialized, the logic of our quiz system is complete. Notice how much easier it is to follow the quiz's operation when most of the code is contained in a single frame. All that's left is to call the *answer( )* function from the appropriate buttons in the quiz.

9.14.1.1 Calling the answer( ) function

We'll call the *answer( )* function from the buttons that the user clicks to indicate answers. Here's the code for the "Answer 1" button:

```
choice1_btn.onRelease = function ( ) {
  this._parent.answer(1);
};
```

This code assigns a function literal to the `choice1_btn` 's *onRelease( )* handler, which is invoked automatically when the button is pressed. The function literal contains a single statement that says, "On the movie clip timeline that contains this button, `this._parent` , invoke *answer( )* , passing the user's choice for this question (1) as an argument: `this._parent.answer(1);` "

Notice that the `choice1_btn` handler applies to any question, so we don't have to assign new button handlers for each question (as we did in the first quiz). The button's job is simply to tell the *answer( )* function which choice the user made. The *answer( )* function keeps track of which question the user is answering, so it records the choice in the appropriate variable.

With the code needed to handle each button neatly packed into the *answer( )* function, our button code is kept to a bare minimum. Furthermore, creating new buttons is a simple matter of passing the correct argument to *answer( )* . To add a fourth choice button, for example, we'd use:

```
choice4_btn.onRelease = function ( ) {
  this._parent.answer(4);
};
```

While that's pretty flexible, we're still duplicating the single statement of the function literal. In the object-oriented version of the quiz, we'll see how to rid our code of this redundancy.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

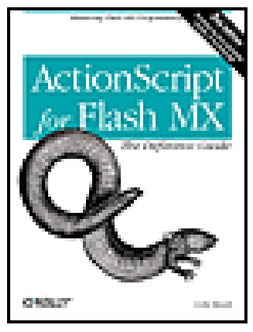
9.14.2 Quiz Summary

By placing all our code on the first frame of our movie, we've separated the *display* of the system from the *logic* of the system. Hence, we don't lose the logic in the maze of frames and buttons that Flash movies can often become.

Test your quiz to make sure that it works. Experiment to see what happens when you add quiz questions, remove quiz questions, and change anything else that comes to mind. As we continue, think about how the techniques you learn might be applied to the quiz. What visible features can we add to the user experience? What invisible changes will make the code more flexible or elegant, even if the user experience is unchanged? Can you alter the quiz to allow more than one correct answer for a question?

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir



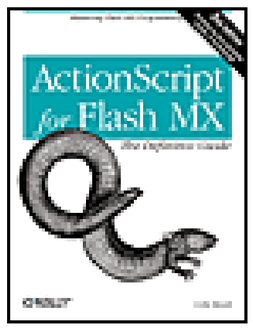
Chapter 9. Functions

9.15 Onward!

This chapter introduced one of the most powerful devices in ActionScript: functions. In the next chapter, you'll learn about a special variety of functions, called *event handlers*, that are executed automatically by the interpreter. Event handlers are one of the key building blocks of interactivity.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir



ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Part I: ActionScript Fundamentals

Chapter 10. Events and Event Handling

You've learned a lot about composing instructions for the ActionScript interpreter to execute. By now you should be pretty comfortable telling the interpreter *what* you want it to do, but how do you tell it *when* to perform those actions? ActionScript code doesn't just execute of its own accord—something always provokes its execution.

That "something" is either the synchronous playback of a movie or the occurrence of a predefined asynchronous *event.*

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Chapter 10. Events and Event Handling

10.1 Synchronous Code Execution

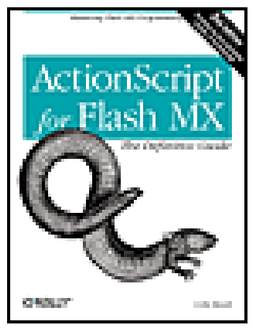
As a movie plays, the timeline's playhead travels from frame to frame. Each time the playhead enters a new frame, the interpreter executes any code attached to that frame. After the code on a frame has been executed, the screen display is updated and sounds are played. Then, the playhead proceeds to the next frame.

For example, when we place code directly on frame 1 of a movie, that code executes before the content in frame 1 is displayed. If we place another block of code on a keyframe at frame 5 of the same movie, frames 1 through 4 will be displayed, then the code on frame 5 will be executed, then frame 5 will be displayed. The code executed on frames 1 and 5 is said to be executed *synchronously* because it happens in a linear, predictable fashion.

All code attached to the frames of a movie is executed synchronously. Even if some frames are played out of order due to a *gotoAndPlay( )* or *gotoAndStop( )* command, the code on each frame is executed in a predictable sequence, synchronized with the movement of the playhead.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir



ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Chapter 10. Events and Event Handling

10.2 Event-Based Asynchronous Code Execution

Some code does not execute in a predetermined sequence. Instead, it executes when the ActionScript interpreter notices that an *event* has occurred. Many events involve a user action, such as clicking the mouse, resizing the movie, or pressing a key. Other events happen without user intervention, such as a sound completing or an XML document loading. Just as the playhead entering a new frame executes synchronous code attached to the frame, events can cause *event-based* code to execute. Event-based code (code that executes in response to an event) is said to be executed *asynchronously* because the triggering of events can occur at arbitrary times.

Synchronous programming requires us to dictate, in advance, the timing of our code's execution. Asynchronous programming, on the other hand, gives us the ability to react dynamically to events as they occur. Asynchronous code execution is critical to ActionScript and interactivity.

This chapter explores asynchronous (event-based) programming in Flash.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Chapter 10. Events and Event Handling

10.3 Types of Events

Conceptually, events can be grouped into two categories:

user events

Actions taken by the user (e.g., a mouseclick or a keystroke)

system events

Things that happen as part of the internal playback of a movie (e.g., a movie clip appearing on stage or variables loading from an external file)

ActionScript does not distinguish syntactically between user events and system events. An event triggered internally by a movie is no less palpable than a user's mouseclick. While we might not normally think of, say, a movie clip's removal from the Stage as a noteworthy "event," being able to react to system events gives us great control over a movie.

ActionScript events can also be categorized more practically according to the object to which they pertain. All events happen relative to some object in the Flash environment. We'll study objects in depth in Chapter 12, but for now, simply assume that content in Flash (buttons, movie clips, text fields, etc) is normally represented as ActionScript objects and that we define events in relation to those objects.

The core ActionScript objects and classes that support events are:

- *Button*
- *Key*
- *LoadVars*
- *LocalConnection*
- *Mouse*
- *MovieClip*
- *Selection*
- *SharedObject*
- *Sound*
- *Stage*

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

- *TextField*
- *XML*
- *XMLSocket*

For details on the specific events supported by these objects, see each object's entry in the *Language Reference.*

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Chapter 10. Events and Event Handling

10.4 Event Handling

Not every event triggers the execution of code or affects a movie. For example, a user can generate dozens of events by clicking repeatedly on a button, but those clicks may be ignored. Why? Because, on their own, events can't cause code to execute—we must write code to react to events explicitly. To instruct the interpreter to execute some code in response to an event, we create either an *event handler* or an *event listener* that describes the action to take when the specified event occurs.

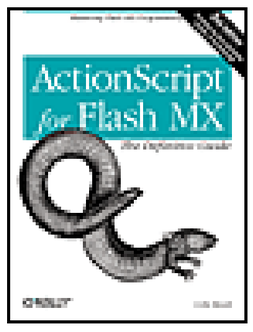
Due to its unusual evolution, ActionScript has four different ways of handling events:

- Event handler properties
- Event listeners
- Button event handlers with the syntax:
```
on (eventName) {
  statements
}
```
- Movie clip event handlers with the syntax:
```
onClipEvent (eventName) {
  statements
}
```

Let's take a look at these event-handling mechanisms individually. Much of the following discussion expects you to understand objects, which are covered in Chapter 12.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir



ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Chapter 10. Events and Event Handling

10.5 Event Handler Properties

Event handlers are so named because they *catch*, or *handle*, the events in a movie. An event handler is simply a function that is invoked automatically when a particular event occurs. We've learned that all events occur in relation to some object. Hence, in order to create an event handler, we must specify three things:

- The object to which the event applies
- The name of the event handler
- The function to be executed when the event occurs (known as the event handler's *callback function*)

For example, suppose we want to animate a movie clip named `ball` by changing its position every time a frame passes in the Flash Player. If we define an *onEnterFrame( )* handler for the `ball` clip, it will be executed once for each tick of the movie's frame rate. So, given a movie clip, `ball`, we can create an *onEnterFrame( )* event handler for `ball` as follows:

1. Create a new empty Flash movie.
2. Create a new movie clip symbol, named *ballSymbol*, that contains a circle shape.
3. Place an instance of *ballSymbol* on *Layer 1*.
4. Name the *ballSymbol* instance `ball`.
5. On frame 1 of *Layer 1* of the main movie timeline, attach the following code:

```
ball.onEnterFrame = moveDown;
function moveDown ( ) {
  // Move the ball down 5 pixels with each frame.
  this._y += 5;
}
```

In this code, `ball` is the object to which the event applies, *onEnterFrame* is the event we wish to handle, and *moveDown* is the function that should run when *onEnterFrame( )* occurs.

Notice that we assign the *moveDown* function reference, not the return value of a *moveDown( )* function call, to *onEnterFrame*. Therefore, the function invocation operator `( )` must not be included.

```
// The following is incorrect! The ( ) operator must not be used.
ball.onEnterFrame = moveDown( );
// The following is correct.
ball.onEnterFrame = moveDown;
```

To be more succinct when assigning an event handler, we can specify the callback function with a function literal, as follows:

```
ball.onEnterFrame = function ( ) {
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

```
  this._y += 5;
};
```

The `ball` example shows how to specify a handler for an clip from outside the clip (i.e., from the main timeline). To define an event handler for the current movie clip from a frame on its own timeline, use the following syntax instead:

```
this.theEventHandler = function ( ) {
  statements
};
```

In general, then, the syntax of an event handler is either:

```
someObject.someEventHandler = function ( ) {
  statements
};
```

in which case the event handler is defined as a function literal, or:

```
someObject.someEventHandler = someFunction;
```

in which case *someFunction* must also be defined elsewhere, as in:

```
function someFunction ( ) {
  statements
}
```

The latter approach is preferred in situations where the event handler is added and removed repeatedly, or where multiple objects use the same function as an event handler.

Event handlers can also receive parameters generated by the event. For example, in the following code, a text field, `userName_txt`, defines an event handler, *onSetFocus( )*, that receives a reference to the previously focused text field via the parameter `oldFocus`. The handler uses `oldFocus` to color the previously focused text field black.

```
userName_txt.onSetFocus = function (oldFocus) {
  // Color the previously focused field black.
  oldFocus.backgroundColor = 0x000000;
  // Color the newly focused field orange.
  this.backgroundColor = 0xF5BF70;
}
```

Object-oriented programmers will recognize event handler syntax as identical to the syntax for defining an object *method*. This is not a matter of coincidence. Event handlers are, in fact, methods that happen to be invoked automatically by the interpreter. And since methods in ActionScript are object properties that store functions, the event handler format shown in our *onSetFocus( )* example is known as an *event handler property*.

Within the body of an event handler callback function, the `this` keyword refers to the object for which the event was triggered. In our `ball`'s *onEnterFrame( )* handler, we used `this` to refer to the `ball` movie clip object, which we moved down by 5 pixels:

```
this._y += 5;
```

Similarly, in our *userName_txt.onSetFocus( )* handler `this` refers to `userName_txt`:

```
  this.backgroundColor = 0xF5BF70;
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

To remove an event handler from an object, use the *delete* operator as follows:

```
delete theObject.theEventHandler;
```

For example, executing the following code from the main timeline causes the `ball` movie clip to stop responding to the *onEnterFrame( )* event:

```
delete ball.onEnterFrame;
```

If the code were attached to the `ball` movie clip itself, you could instead write:

```
delete this.onEnterFrame;
```

For a list of event handlers offered by a specific class or object, see the appropriate entry in the *Language Reference.* In particular, be sure to study the *Button* and *MovieClip* classes, which define the majority of user-interaction events. For example, to cause a *submitForm( )* function to execute when a Submit button is pressed, we use:

```
submit_btn.onRelease = submitForm;
function submitForm ( ) {
  // Validate data and send it to the server...
}
```

where *submit_btn* is the button object, the *onRelease* event is invoked when the button is pressed and then released, and *submitForm( )* is a custom function that sends the form data to the server. For a complete form-submission example, see Chapter 17.

Event handlers are convenient, but they also suffer from a shortcoming: only one callback function can be *registered* (assigned) to an event handler at a time. That is, only one action can be performed for each event when using event handler properties. We'll learn how to overcome this limitation in the next section using *listener events.*

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Chapter 10. Events and Event Handling

10.6 Listener Events

Conceptually, *listener events* are similar to event handler properties. Like event handler properties, listener events are handled with a specially named object method (i.e., a property containing a callback function) that is invoked when a given event occurs. However, listener events add two important event handling features:

- An object can respond to events that occur for other objects, not just itself.
- More than one object can respond to an event.

For example, suppose we want to center a text field, `title_txt`, in a movie by setting its position each time the movie changes size. Consulting the *Language Reference*, we determine that when a movie changes size the *Stage.onResize( )* listener event is triggered. Our text field is not an instance of *Stage*, yet we want it to respond to *Stage.onResize( )*. This is no problem. Because *State.onResize( )* is a listener event, it will let `title_txt` (and any other object) sign up to be notified when *onResize( )* occurs.

Here's how it works. First, we create our *onResize( )* method directly on `title_txt`, just as we created our earlier event handler properties:

```
title_txt.onResize = function ( ) {
  // Center the text field.
  // Stage.width and Stage.height always store the current movie dimensions.
  // In the callback function body we refer to title_txt as this.
  this._x = Stage.width/2 - this._width/2;
  this._y = Stage.height/2 - this._height/2;
}
```

Next, we must tell the *Stage* object that we want it to execute `title_txt`'s *onResize( )* method when the *onResize( )* event occurs. This step is known as "adding a listener." The so-called *listener* is the object (in our case, `title_txt`) that wishes to receive event notifications from the *event source* or, synonymously, the *event broadcaster* (in our case, *Stage*). All listener event sources provide a special method, *addListener( )*, that is used to register new listeners. For example, here's how we add `title_txt` as a *Stage* listener:

```
Stage.addListener(title_txt);
```

Henceforth, when the *Stage.onResize( )* event occurs, *title_txt.onResize( )* is invoked. In fact, as a *Stage* listener, `title_txt` will receive event notices for all of *Stage*'s listener events (though in *Stage's case*, it defines only one listener event). Some event sources define both event handler properties and listener events. In such a case, a listening object will receive notifications for the listener events only, not the event handler properties. Each object's supported listener events and event handler properties are listed in the ActionScript Language Reference.

Generally, an event is implemented as a handler property when it is of interest only to the object that generates it. Listener events are more common when many different kinds of objects have interest in a single event (for example, a movie clip,

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

a button, and a text field may all want to know when a movie is resized, so *Stage.onResize( )* is a listener event).

Here is the general syntax for creating an *event listener* (an object that defines a method that handles a listener event):

```
listenerObject.eventName = function ( ) {
  statements
}
eventSource.addListener(listenerObject);
```

The `listenerObject` can be any object. The `eventName` is the name of the predefined event to listen for (e.g., *onResize* in *title_txt.onResize( )*). The `eventSource` is the object that notifies `listenerObject` when `eventName` occurs. Therefore, unless `eventName` is one of the events broadcast by `eventSource`, the event listener method will never be executed. For details on specific event sources see the *Key*, *Mouse*, *Selection*, *TextField*, and *Stage* objects in the *Language Reference*.

Getting back to our example, suppose we want to center a second text field, `subtitle_txt`, underneath `title_txt`. We follow the same process we did with `title_txt`. First, we create our *onResize( )* method directly on `subtitle_txt`:

```
subtitle_txt.onResize = function ( ) {
  this._x = Stage.width/2 - this._width/2;
  this._y = (Stage.height/2) + (title_txt._height/2) + 10;
}
```

Then, we register `subtitle_txt` as a *Stage* listener:

```
Stage.addListener(subtitle_txt);
```

Note that the *onResize( )* methods must be written appropriately for each object that we wish to reposition. In this case, we reposition `title_txt` based on the height and width of the Stage and the text object itself. We offset `subtitle_txt` vertically, relative to the center of the Stage and `title_txt`'s height.

> In order to function properly, the *Stage.onResize( )* event requires `Stage.scaleMode` and `Stage.align` to be set, as shown in Example 10-1.

Adding a listener to an event source is sometimes referred to as *subscribing* the listener. We now have two text fields—each with their own independent *onResize( )* method—that respond to a single event, *Stage.onResize( )*. We can also register movie clips, buttons, or any other objects as *Stage* listeners, allowing for a entire movie's layout to adjust dynamically when the movie is resized. When multiple listeners are registered to a single event source, their methods are executed in the order the listeners were added.

To remove (or *unsubscribe*) any listener from an event source, use the *removeListener( )* method. For example, the following code causes `subtitle_txt` to stop receiving *onResize* events, so it won't reposition itself when the movie is resized:

```
Stage.removeListener(subtitle_txt);
```

The general form to remove a listener is:

```
eventSource.removeListener(listenerObject);
```

Example 10-1 shows our text field centering code in its entirety. To test it, attach the code to the first frame of the main

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

timeline in a new movie; then export the movie and try resizing the movie's window. For complete information on building resizable interfaces, see the *Stage* object in the ActionScript Language Reference.

Example 10-1. Keeping two text fields centered in a movie

```
// Set the scaling behavior of the movie
Stage.scaleMode = "noScale";
Stage.align = "LT";

// Create the first text field
this.createTextField("title_txt", 1, 0, 0, 180, 20);
title_txt.text = "Welcome to my website.";
title_txt.border = true;

// Assign its listener callback function
title_txt.onResize = function ( ) {
  this._x = Stage.width/2 - this._width/2;
  this._y = Stage.height/2 - this._height/2;
}

// Create the second text field
this.createTextField("subtitle_txt", 2, 0, 0, 180, 20);
subtitle_txt.text = "Thanks for visiting.";
subtitle_txt.border = true;

// Assign its listener callback function
subtitle_txt.onResize = function ( ) {
  this._x = Stage.width/2 - this._width/2;
  this._y = (Stage.height/2) + (title_txt._height/2) + 10;
}

// Register the text fields as Stage listeners
Stage.addListener(subtitle_txt);
Stage.addListener(title_txt);
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Chapter 10. Events and Event Handling

10.7 Flash 5's on( ) and onClipEvent( ) Handlers

Earlier in this chapter, we discussed how to assign callback functions to the event handler properties *MovieClip.onEnterFram* and *TextField.onSetFocus( )* . In Flash 5, these kinds of event handler properties were available on *XML* and *XMLSocket* or Movie clips and buttons used idiosyncratic event handler syntax attached directly to physical clips or buttons in the Flash authoring tool. Button event handlers were defined using *on(* `eventName` *)* , and movie clip event handlers were defined using *onClipEvent(* `eventName` *)* , where `eventName` specified the event to be handled. As of Flash MX, use of these older forms is generally discouraged. Event handler properties and event listeners are preferred because they promote the separation of code from the visual elements of a movie. However, the older syntax is still supported, and there are a few speci cases where it is actually required (for examples, see *MovieClip.onLoad( )* and *Button keyPress* in the *Language Reference*).

Flash 5-style button *on( )* event handlers take the form:

```
on (eventName) {
  statements
}
```

The `eventName` is the name of the corresponding Flash MX event handler property, without the word "on" and with the fi letter lowercase. For example, the *onRelease( )* property in Flash MX was implemented as follows in Flash 5:

```
on (release) {
  statements
}
```

When exporting to Flash 5 format, you must use the Flash 5-style event syntax. A single button handler can respond to multip events, separated by commas. For example:

```
on (rollOver, rollOut) {
  // Invoke a custom function in response to both the rollOver and rollOut eve
  playRandomSound( );
}
```

To upgrade such code to Flash MX's preferred syntax, using the same callback handler to respond to both events, you'd use:

```
theButton.onRollOver = playRandomSound;
theButton.onRollOut = playRandomSound;
```

or, even more tersely:

```
theButton.onRollOver = theButton.onRollOut = playRandomSound;
```

Flash 5-style movie clip *onClipEvent( )* handlers take the form:

```
onClipEvent (eventName) {
  statements
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

```
}
```

Again, the *eventName* is the name of the corresponding Flash MX event handler property, with the word "on" removed. F example, the Flash MX *MovieClip.onEnterFrame( )* event handler was implemented as follows in Flash 5:

```
onClipEvent (enterFrame) {
  statements
}
```

Unlike button handlers, clip handlers can respond to a single event only.

Events introduced after Flash 5 are not available in *on( )* or *onClipEvent( )* form. For example, there is no such thing as *onClipEvent(setFocus)* , because the *onSetFocus( )* event was introduced in Flash MX and first supported by Flash Player 6.

As of Flash MX, movie clip instances can take both *onClipEvent( )* and button-style *on( )* event handlers. For details, see Sec 13.9 in Chapter 13.

10.7.1 Attaching Event Handlers to Buttons and Movie Clips

To attach a Flash 5-style *on( )* or *onClipEvent( )* event handler to a button or a movie clip, we must physically place the code the handler onto the desired button or clip. We can do so in the Flash authoring tool only, by selecting the object on stage and entering the appropriate code in the Actions panel, as shown in Figure 10-1.

Figure 10-1. Attaching an event handler to a button

Let's try making two simple event handler functions, one for a button and another for a movie clip. To create a button event handler, follow these instructions:

1. Create a new, blank Flash movie.
2. Create a button and drag an instance of it onto the main Stage.
3. With the button selected, type the following code in the Actions panel:

```
on (release) {
  trace("You clicked the button");
}
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

4. Select Control → Test Movie.
5. Click the button. The message, "You clicked the button", appears in the Output window.

When the movie plays and we click and release the mouse over the button, the *release* event is detected by the interpreter, wh executes the *on(release)* event handler. Each time we press and release the mouse, the message, "You clicked the button", app in the Output window.

Now let's try making a slightly more interesting event handler on a movie clip.

1. Create a new, blank Flash movie.
2. On the main movie Stage, draw a rectangle.
3. Select Insert → Convert to Symbol.
4. In the Convert to Symbol dialog box, name the new symbol *rectangle* , and select Movie Clip as the Behavior.
5. Click OK to finish creating the `rectangle` movie clip.
6. Select the `rectangle` clip on stage, and then type the following in the Actions panel:

```
onClipEvent (keyDown) {
  this._visible = false;
}
onClipEvent (keyUp) {
  this._visible = true;
}
```

7. Select Control Test Movie.
8. Click the movie to make sure it has keyboard focus, then press and hold any key. Each time you depress a key, the `rectangle` movie clip disappears. Each time you release the depressed key, `rectangle` reappears.

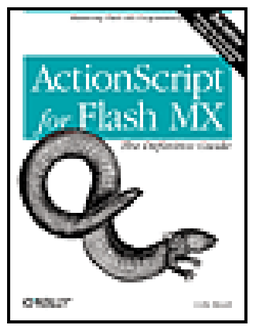
Unlike Flash 6's event handler properties and event listeners, Flash 5's *on( )* and *onClipEvent( )* handlers cannot be attached a removed via ActionScript while the movie is playing. Therefore, in Flash 5 the following event handler property assignment i illegal:

```
theClip.onKeyDown = function ( ) {
  _visible = 0;
};
```

We'll see how to work around this shortcoming later in this chapter under Section 10.15.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir



ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Chapter 10. Events and Event Handling

10.8 Event Handler Lifespan

The lifespan of event handlers is tied to the life of the objects with which they are associated. When a clip or button is removed from the Stage, or when an *XML* or *TextField* object is deleted, any event handlers associated with those objects die with them. An object must be present on stage or in code for its handlers to remain active.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Chapter 10. Events and Event Handling

10.9 Event Handler Scope

As with any function, the statements in an event handler or listener execute within a predefined scope. Scope dictates where t interpreter looks to resolve unqualified variable references in an event handler's body. We'll consider event handler scope in relation to the four event implementations supported by Flash.

10.9.1 Scope of Event Handler Properties and Event Listener Methods

Event listener methods and event handler properties are scoped exactly as functions are scoped. An event callback function ha a scope chain that is defined when the function is defined. Furthermore, event handler properties and event listener methods define a local scope. All the rules of function scope described in Chapter 9 apply to event handler properties and event listene methods.

10.9.2 Scope of onClipEvent( ) Handlers

Unlike event handler properties and event listener methods, *on( )* and *onClipEvent( )* handlers *do not define a local scope* ! When we attach an *onClipEvent( )* handler to a clip, the scope of the handler is the clip, not the event handler. This means tha all variables are retrieved from the clip's timeline. For example, if we attach an *onClipEvent(enterFrame)* handler to a clip named `navigation` and write *trace(x);* inside the handler, the interpreter looks for the value of `x` on `navigation` 's timeline:

```
onClipEvent (enterFrame) {
  trace(x);  // Displays the value of navigation.x
}
```

The interpreter does not consult a local scope first, because there is no local scope to consult. If we write *var y = 10;* in the *onClipEvent( )* handler, `y` is defined on `navigation` 's timeline, even though the *var* keyword ordinarily declares a local variable when used in a function.

The easiest way to remember the scope rules of an *onClipEvent( )* handler is to treat the handler's statements as though they were attached to a frame of the handler's clip. For example, suppose we have a clip named `ball` on the main timeline that ha a variable called `xVelocity` in it. To access `xVelocity` from inside `ball` 's *onClipEvent( )* handler, we simply refer t it directly, like this:

```
onClipEvent (mouseDown) {
  xVelocity += 10;
}
```

Supplying the path to the variable as `_root.ball.xVelocity` is optional, because the interpreter already assumes we mean the variable `xVelocity` in `ball` . The same is true of properties and methods; instead of using `ball._x` , we can use `_x` , and instead of using *ball.gotoAndStop(5)* , we can use *gotoAndStop(5)* . For example:

```
onClipEvent (enterFrame) {
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

```
  _x += xVelocity;                    // Move the ball
  gotoAndPlay(_currentframe - 1);  // Do a little loop
}
```

We can even define a function on `ball` using a function declaration statement in an *onClipEvent( )* handler, like this:

```
onClipEvent (load) {
  function hideMe () {
    _visibility = 0;
  }
}
```

However, unqualified variable references are ambiguous and behave differently in an event handler property or event listener method than they do in an *onClipEvent( )* handler. Therefore, it's prudent to always use the `this` keyword when referring to the movie clip that defines the handler:

```
// Preferred reference to the clip with the onClipEvent( ) handler:
this._x
// Legal, but discouraged reference to the clip with the onClipEvent( ) handler
_x

// Preferred:
this.gotoAndStop(12);
// Discouraged:
gotoAndStop(12);
```

Additionally, use of `this` is required when we're dynamically generating the name of one of the current clip's properties (either a variable name or a nested clip). Here we tell one of the nested clips in the series `ball.stripe1` , `ball.stripe2` , etc. to start playing, depending on the current frame of the `ball` clip:

```
onClipEvent (enterFrame) {
  this["stripe" + this._currentframe].play();
}
```

The `this` keyword should also be used with movie clip methods that demand an explicit reference to a movie clip object upon invocation. Any movie clip method with the same name as an ActionScript global function must be used with an explicit clip reference. The `this` keyword is therefore necessary when invoking the following functions as methods inside an *onClipEvent( )* handler:

duplicateMovieClip ( )
loadMovie ( )
loadVariables ( )
removeMovieClip ( )
startDrag ( )
unloadMovie ( )

For example:

```
this.duplicateMovieClip("ball2", 1);
this.loadVariables("vars.txt");
this.startDrag(true);
this.unloadMovie();
```

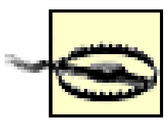
downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

You'll learn all about the dual nature of these functions in Chapter 13 under Section 13.7.

Note that the `this` keyword allows us to refer to the current clip even when that clip has no assigned instance name in the authoring tool or when we don't know the clip's name. In fact, using `this` , we can even pass the current clip as a reference t a function without ever knowing the current clip's name. Here's some code to demonstrate:

```
// CODE ON MAIN TIMELINE
// Here is a generic function that moves any clip
function move (clip, x, y) {
  clip._x += x;
  clip._ y += y;
}

// CODE ON CLIP
// Call the main timeline function and tell it to move the
// current clip by passing a reference with the this keyword
onClipEvent (enterFrame) {
  _root.move(this, 10, 15);
}
```



In Flash Player 5.0.30.0, a bug prevents *gotoAndStop( )* and *gotoAndPlay( )* from working inside a clip handler when used with string literal labels. Such commands are simply ignored. For example, this does not work:

```
onClipEvent (load) {
   gotoAndStop("intro"); // Won't work in Flash 5.0.30.0
}
```

To work around the bug, use a self-reflexive clip reference (i.e., `this`), as in:

```
onClipEvent(load) {
   this.gotoAndStop("intro");
}
```

Remember that statements in *onClipEvent( )* handlers are scoped to the *clip's* timeline, not to the handler function's local scope or the clip's *parent* timeline (the timeline upon which the clip resides).

For example, suppose we place our `ball` clip on the main timeline of a movie, and the main timeline (not `ball` 's timeline) has a *moveBall( )* function defined on it. We may absent-mindedly call *moveBall( )* from an event handler on `ball` , like thi

```
onClipEvent (enterFrame) {
  moveBall( );  // Does nothing! There's no moveBall( ) function in ball.
                // The moveBall( ) function is defined on _root.
}
```

We must refer to the *moveBall( )* function explicitly on the main timeline using `_root` like this:

```
onClipEvent (enterFrame) {
  _root.moveBall( );  // Now it works!
}
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

10.9.3 Scope of on( ) Handlers

The scope of an *on( )* handler depends on the type of the object to which it is attached. When attached to a button, an *on( )* handler is scoped to the timeline upon which the button resides. When attached to a movie clip, an *on( )* handler is scoped to t clip that bears the handler, as with an *onClipEvent( )* handler. Prior to Flash MX, movie clips did not support *on( )* handlers (only Flash 5 buttons did).

For example, if we place a button on the main timeline and declare the variable `speed` in a handler on that button, `speed` will be scoped to the main timeline (`_root`):

```
// CODE FOR BUTTON HANDLER
on (release) {
  var speed = 10;  // Defines speed on _root
}
```

In contrast, if we place an *on( )* handler on a movie clip object named `ball` and declare the variable `speed` in the handler, `speed` is scoped to `ball` :

```
// CODE FOR ball HANDLER
on (release) {
  var speed = 10;  // Defines speed on ball, not _root
}
```

Inside an *on( )* handler for a button object, the `this` keyword refers to the timeline on which the button resides:

```
// CODE FOR BUTTON HANDLER
on (release) {
  // Make the clip on which this button resides 50% transparent
  this._alpha = 50;
  // Move the clip on which this button resides 10 pixels to the right
  this._x += 10;
}
```

But inside a movie clip's *on( )* handler, the `this` keyword refers to the clip that bears the handler:

```
// CODE FOR MOVIE CLIP HANDLER
on (release) {
  // Make the clip with the handler 50% transparent
  this._alpha = 50;
  // Move the clip with the handler 10 pixels to the right
  this._x += 10;
}
```

See the *MovieClip* and *Button* classes in the *Language Reference* for details on movie clip and button events.

10.9.4 Scope Summary

The various scopes in the previous discussion can be confusing. Use the general rules outlined in Table 10-1 to determine the scope of a specific handler in Flash MX. Note that of the following event handling mechanisms, Flash 5 supported only "MovieClip with *onClipEvent( )* handler" and "Button with *on( )* handler."

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Table 10-1. Event scope summary

Event handling mechanism	Example	Scope

Event handler property	`someTextField.onSetFocus = function ( ) {`
	`  statements`
	`}`
		The clip in which function declaration occurred.[1]

Event listener method	`GUIManager.onResize = function ( ) {`
	`  statements`
	`}`
	`Stage.addListener(GUIManager);`
		The clip in which function declaration occurred.[1]

MovieClip with event handler property	`theClip_mc.onPress = function ( ) {`
	`  statements`
	`}`
		The clip in which function declaration occurred.[1]

MovieClip with *on( )* handler	`on (press) {`
	`  statements`
	`}`
		The clip that bears the handler in the authoring tool.

MovieClip with *onClipEvent( )* handler	`onClipEvent (enterFrame) {`
	`  statements`
	`}`
		The clip that bears the handler in the authoring tool.

Button with event handler property	`submit_btn.onRelease = function ( ) {`
	`  statements`
	`}`
		The clip in which function declaration occurred.[1]

Button with *on( )* handler	`on (release) {`
	`  statements`
	`}`
		The clip on whose timeline the button resides.

[1] For handlers and listeners assigned within a function, the scope is that function (in accordance with the rules of nested function scope, discussed in Chapter 9).

Finally, here's a scenario to test your understanding of event handler scope. Before reading the answer, see if you can determi it yourself.

Suppose a movie clip instance named `intro_mc` is placed on the main timeline of a movie, and the following code is attached to the main timeline:

```
intro_mc.onPress = function ( ) {
  trace(this);  // Here, this refers to intro_mc.
  this.play( )   // This line makes intro_mc play.
  play( );       // Why won't this line make intro_mc play?
}
```

Question: In `intro_mc` 's *onPress( )* callback function, why does `this` refer to `intro_mc` , whereas unqualified statements do not? In other words, why doesn't line 3 of the handler body make `intro_mc` play?

Answer: `this` is a reference to the object through which *onPress( )* was invoked: `intro_mc` . But statements inside the *onPress( )* function are executed in the scope of the timeline on which the function was defined—the main timeline. Hence, th standalone *play( )* command plays the main timeline, not `intro_mc` .

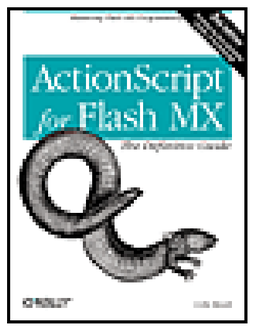
downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Moral of the story: obviously, the location of the function declaration can change, so it's a little dangerous to use unqualified references inside handlers at all. Qualify all nonlocal identifiers within handlers with `this` or some other explicit object reference.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir



ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Chapter 10. Events and Event Handling

10.10 Values of the this Keyword

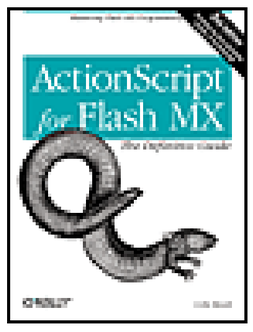
In our study of ActionScript's event handling tools, we've encountered the `this` keyword several times. Within an event handler, `this` normally refers to the object that received notification of the event. However, Flash's evolution through multiple event systems has complicated the issue. Table 10-2 provides a guide to the values of the `this` keyword in ActionScript's various event handling mechanisms.

Table 10-2. Values of the this keyword

Event handling mechanism	Value of this
Event handler property	The object that defines the handler property.
Event listener method	The object registered as a listener.
MovieClip with event handler property	The movie clip that defines the handler property.
MovieClip with *on( )* handler	The clip that bears the handler in the authoring tool.
MovieClip with *onClipEvent( )* handler	The clip that bears the handler in the authoring tool.
Button with event handler property	The button that defines the handler property.
Button with *on( )* handler	The clip on whose timeline the button resides.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir



Chapter 10. Events and Event Handling

10.11 Flash 5-style onClipEvent( ) Order of Execution

Some movies have code dispersed across multiple timelines and *on( )* and *onClipEvent( )* handlers. It's not uncommon, therefore, for a single frame to require the execution of many separate blocks of code—some in event handlers, some on frames in clip timelines, and some on the main timelines of documents in the Player. In these situations, the order in which the various bits of code execute can become quite complex and can greatly affect a program's behavior. We can prevent surprises and guarantee that our code behaves as desired by becoming familiar with the order in which event handlers execute, relative to the various timelines in a movie.

User-input event handlers execute independently of the code on a movie's timelines. Button event handlers, for example, are executed immediately when the event that they handle occurs, as are handlers for the *onMouseDown( )*, *onMouseUp( )*, *onMouseMove( )*, *onKeyDown( )*, and *onKeyUp( )* events.

Handlers for *onClipEvent( )* events, however, execute in order, according to the progression of the movie, as shown in Table 10-3.

Table 10-3. Flash 5-style onClipEvent( ) order of execution

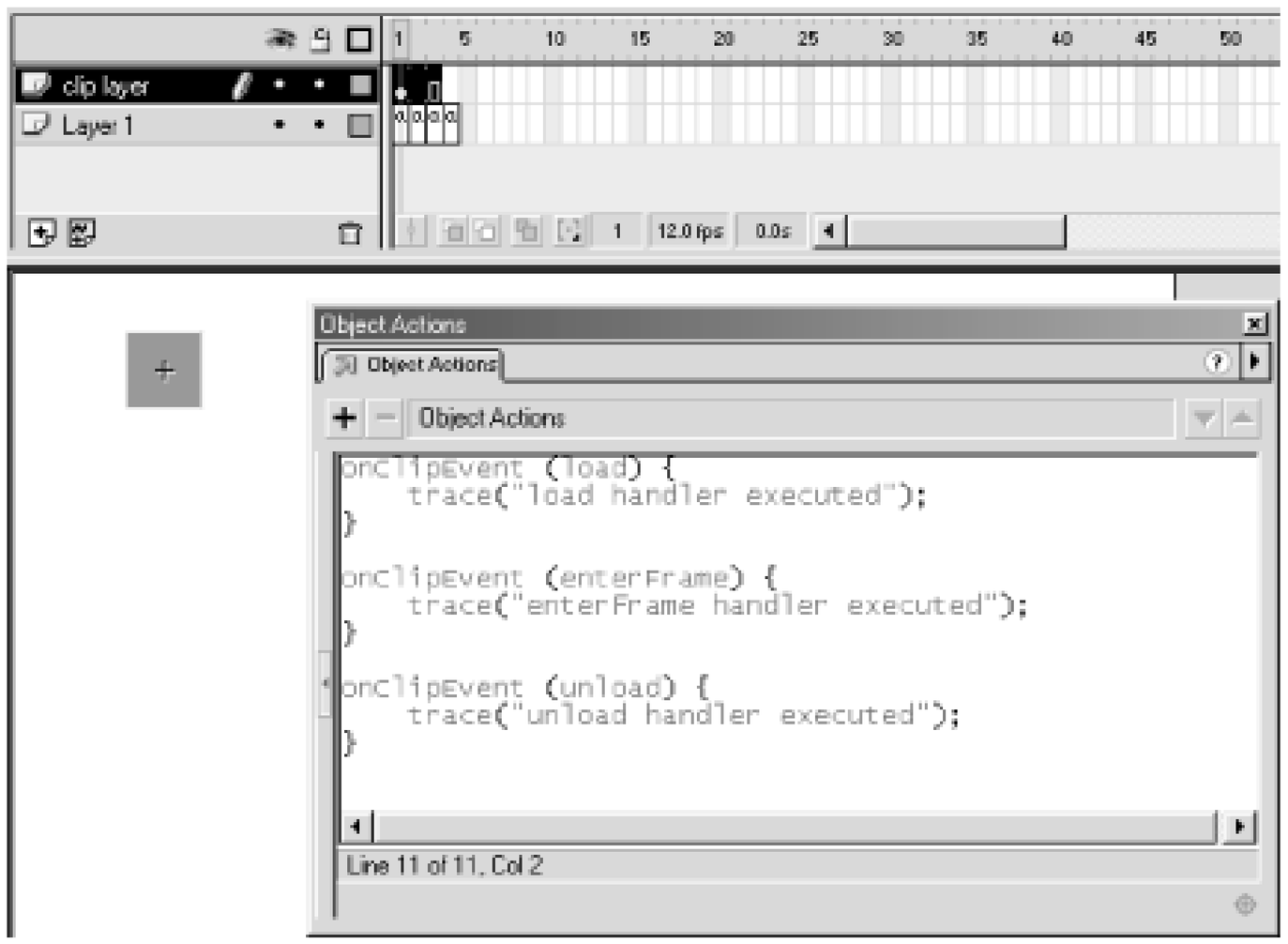
Event handler	Execution timing
load	Executes in the first frame in which the clip is present on stage after parent-timeline code executes, but before clip-internal code executes, and before the frame is rendered. See special notes listed under *MovieClip.onLoad( )* in the ActionScript Language Reference.
unload	Executes in the first frame in which the clip is not present on stage, before parent-timeline code executes.
enterFrame	Executes in the second frame and all subsequent frames in which the clip is present on stage. It is executed before parent-timeline code executes and before clip-internal code executes.
data	Executes in any frame in which data is received by the clip. If triggered, it executes before clip-internal code executes and before *enterFrame* code executes.

It's easier to see the effect of the rules in Table 10-3 with a practical example. Suppose we have a single-layer movie with four keyframes in the main timeline. We attach some code to each keyframe. Then, we create a second layer where we place a movie clip at frame 1, spanning to frame 3, but not present on frame 4. We add *load*, *enterFrame*, and *unload onClipEvent( )* handlers to our clip. Finally, inside the clip, we create three keyframes, each of which also contains a block of code. Figure 10-2 shows what the movie looks like.

Figure 10-2. A code execution order test movie

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir



When we play our movie, the execution order is as follows:

```
=  =  =  =  =  =  =  =FRAME 1=  =  =  =  =  =  =
1) Main timeline code executed
2) load handler executed
3) Clip-internal code, frame 1, executed

=  =  =  =  =  =  =  =FRAME 2=  =  =  =  =  =  =
1) enterFrame handler executed
2) Clip-internal code, frame 2, executed
3) Main timeline code executed

=  =  =  =  =  =  =  =FRAME 3=  =  =  =  =  =  =
1) enterFrame handler executed
2) Clip-internal code, frame 3, executed
3) Main timeline code executed

=  =  =  =  =  =  =  =FRAME 4=  =  =  =  =  =  =
1) unload handler executed
2) Main timeline code executed
```

The execution order of the code in our sample movie demonstrates some important rules of thumb to remember when coding with movie clip event handlers:

- Code in a *load* handler is executed before internal clip code, so a *load* handler can be used to initialize variables that are used immediately on frame 1 of its associated clip, but only if the handler is created with the *onClipEvent( )* syntax. See *MovieClip.onLoad( )* in the *Language Reference* for further details.

- Before a movie clip is instantiated on a frame, the code of that frame is executed. Therefore, user-defined variables and functions in a movie clip are not available to any code on its parent timeline until the frame *after*

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

the clip first appears on stage, even if those variables and functions are declared in the clip's *load* handler. For a solution to this problem, see `#initclip` in the ActionScript Language Reference.

- The *enterFrame* event never occurs on the same frame as the *load* or the *unload* event. The *load* and *unload* events supplant *enterFrame* for the frames where a clip appears on the Stage or leaves the Stage.

- On each frame, code in a clip's *enterFrame* handler is executed before code on the clip's parent timeline. Therefore, using an *enterFrame* handler, we can change the properties of a clip's parent timeline and then immediately use the new values in that timeline's code, all on the same frame.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Chapter 10. Events and Event Handling

10.12 Copying Movie Clip Event Handlers

Here's a quick point that has important ramifications: *onClipEvent( )* and *on( )* handlers are duplicated when a movie clip is duplicated via the *duplicateMovieClip( )* function. However, event handler properties, such as *onEnterFrame( )*, are not! Suppose, for example, we have a movie clip on stage named `square`, which has an *onClipEvent(load)* handler defined:

```
onClipEvent (load) {
  trace("movie loaded");
}
```

What happens when we duplicate `square` to create `square2`?

```
square.duplicateMovieClip("square2", 0);
```

Answer: Because the *onClipEvent(load)* handler is copied to `square2` when we duplicate `square`, the birth of `square2` causes its *load* handler to execute, which displays "movie loaded" in the Output window. By using this automatic retention of handlers, we can create slick recursive functions with very powerful results. For a demonstration, refer to *MovieClip.onLoad( )* in the ActionScript Language Reference.

In contrast, suppose we have a movie clip named `circle` and we assign it an *onEnterFrame( )* event handler property as follows:

```
circle.onEnterFrame = function ( ) {
  trace(this._name);
}
```

What happens when we duplicate `circle` to create `circle2`?

```
circle.duplicateMovieClip("circle2", 1);
```

Answer: Only `circle`'s name appears in the Output window. The *onEnterFrame( )* event handler property is not copied to `circle2`, so `circle2`'s name does not appear in the Output window. To force every duplicate of the `circle` clip to define an *onEnterFrame( )* event handler property, we can set the property on a frame of `circle`'s timeline:

```
this.onEnterFrame = function ( ) {
  trace(this._name);
}
```

Even more eloquently, we can define `circle` as a *MovieClip* subclass, and define the *onEnterFrame( )* handler on the `circle` constructor's `prototype`. There's much more to come on that topic in Chapter 14.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Chapter 10. Events and Event Handling

10.13 Refreshing the Screen with updateAfterEvent( )

The *MovieClip* event handler properties *onMouseDown( )*, *onMouseUp( )*, *onMouseMove( )*, *onKeyDown( )*, and *onKeyUp( )* are executed immediately upon the occurrence of their corresponding events. Immediately means *immediately*—even if the event in question occurs between the rendering of frames.

This immediacy can give a movie great responsiveness, but that responsiveness can easily be lost by improper coding. By default, the visual effects of *onMouseDown( )*, *onMouseUp( )*, *onMouseMove( )*, *onKeyDown( )*, and *onKeyUp( )* are not physically rendered by the Flash Player until the next available frame is rendered. To see this in action, create a single-frame movie with a frame rate of 1 frame per second. Place a movie clip named `circle_mc` on the Stage and attach the following code to frame 1 of the main timeline:

```
circle_mc.onMouseDown = function ( ) {
  this._x += 10;
}
```

Then, test the movie and click the mouse as fast as you can. You'll see that all your clicks are registered, but the movie clip still moves only once per second. So, if you click 6 times between frames, the clip will move 60 pixels to the right when the next frame is rendered. If you click 3 times, the clip will move 30 pixels. The cumulative effect of each execution of the *onMouseDown( )* event is "remembered" between frames, but the results are displayed only when each frame is rendered. This can have dramatic effects on certain forms of interactivity.

Fortunately, we can force Flash to immediately render any visual change that takes place during a user-input event handler, without waiting for the next frame to come around. We simply use the *updateAfterEvent( )* function from inside our event handler, like this:

```
circle_mc.onMouseDown = function ( ) {
  this._x += 10;
  updateAfterEvent( );
}
```

The *updateAfterEvent( )* function is available for use only with the *onMouseDown( )*, *onMouseUp( )*, *onMouseMove( )*, *onKeyDown( )*, and *onKeyUp( ) MovieClip* events and the *setInterval( )* function. It is often essential for smooth and responsive visual behavior associated with user input. Later, in Example 10-3, we'll use *updateAfterEvent( )* to ensure the smooth rendering of a custom pointer. Note, however, that button events do not require an explicit *updateAfterEvent( )* function call. Button events update the Stage automatically between frames.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

The *onMouseDown( )*, *onMouseMove( )*, and *onMouseUp( )* listeners were added to the *Mouse* object in Flash Player 6. Likewise, the *onKeyDown( )* and *onKeyUp( )* listeners were added to the *Key* object. Generally, these *Mouse* and *Key* listeners are preferred over the analogous *MovieClip* events. However, in Flash Player 6, the *updateAfterEvent( )* function can be used only in *MovieClip* events. Hence, when a screen refresh is required between frames, the *MovieClip* event handlers must be used. Future versions of the Player will likely include support for *updateAfterEvent( )* from *Mouse* and *Key* listeners.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Chapter 10. Events and Event Handling

10.14 Code Reusability

When using event handlers and listeners, don't forget the code-centralization principles discussed in Chapter 9. Avoid unnecessary duplication and intermingling of code across movie elements. If you find yourself entering the same code in more than one event handler's body, you should try defining it in a single function and pointing each event handler to that function. Try generalizing your code, pulling it off the object and placing it in a code repository somewhere in your movie; global code used throughout a movie should be attached to the `_global` object.

In almost all cases, it's a poor idea to hide statements inside an *on( )* or *onClipEvent( )* handler. Remember that encapsulating your code in a function makes your code reusable and easy to find. This is particularly true of buttons—I rarely place any code on a button in Flash MX, and I rarely use more than a single function-invocation statement in a Flash 5 *on( )* handler. For movie clips, you'll need to employ keener judgment, as placing code directly on clips can often be a healthy part of a clean, self-contained code architecture. Experiment with different approaches until you find the right balance for your needs and skill level. Regardless, it always pays to be mindful of redundancy and reusability issues.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Chapter 10. Events and Event Handling

10.15 Dynamic Movie Clip Event Handlers

Earlier in this chapter, we saw that Flash 5-style *on( )* and *onClipEvent( )* handlers cannot be changed or removed during movie playback. Furthermore, *onClipEvent( )* event handlers cannot be attached to the main movie timeline (the `_root`) of any movie.

In order to work around these limitations, we can—in the case of *onClipEvent(enterFrame)* and the movie clip mouse and key events—use empty movie clips to simulate dynamic event-handler removal and alteration. Empty movie clips even let us simulate *onClipEvent( )* handlers for the main timeline. Follow these steps to see how it works (note that the following technique applies to Flash 5 only, because event handler properties in Flash MX can be removed directly via *delete*):

1. Create an empty movie clip named `process`.
2. Place another empty clip, named `eventClip`, inside `process`.
3. On `eventClip`, attach the desired event handler. The code in the `eventClip`'s handler should target the `process` clip's host timeline, like this:

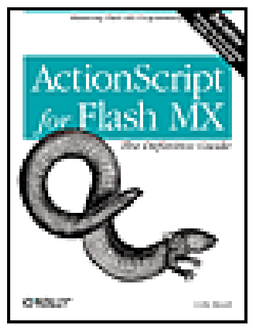
```
onClipEvent (mouseMove) {
  this._parent._parent.doSomeFunction( );
}
```

4. To export `process` for use with the *attachMovie( )* function, select it in the Library and choose Options Linkage. Set Linkage to Export This Symbol, and assign an appropriate identifier (e.g., *mouseMoveProcess*).
5. Finally, to engage the event handler, attach the `process` clip to the appropriate timeline using *attachMovie( )*.
6. To disengage the handler, remove the `process` clip using *removeMovieClip( )*.

To see this technique in action, download "Event Loop, Controllable" from the online Code Depot.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir



ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Chapter 10. Events and Event Handling

10.16 Event Handlers Applied

We'll conclude our exploration of ActionScript events and event handlers with a couple of real-world examples. These are simple applications, but they give us a sense of how flexible event-based programming can be. For more event code samples, see the ActionScript Language Reference coverage of the events associated with the objects listed at the beginning of this chapter.

Example 10-2 makes a clip named `box_mc` shrink and grow.

Example 10-2. Oscillating the size of a movie clip

```
// Set oscillation parameters
box_mc.sizeIncrement = 10;
box_mc.maxHeight = 200;
box_mc.minHeight = 20;

// Size the box with each passing frame
box_mc.onEnterFrame = function ( ) {
  if (this._height >= this.maxHeight || this._height <= this.minHeight) {
    this.sizeIncrement = -this.sizeIncrement;
  }
  this._height += this.sizeIncrement;
  this._width  += this.sizeIncrement;
}
```

Example 10-3 simulates a custom mouse pointer by hiding the normal system pointer and making a clip follow the mouse location around the screen. In the example, the *onMouseDown( )* and *onMouseUp( )* handlers resize the custom pointer slightly to indicate mouseclicks. The code is placed inside the clip that acts as the custom pointer.

Example 10-3. A custom mouse pointer

```
Mouse.hide( );

this.onMouseMove = function ( ) {
  // When the mouse moves, position the current clip at
  // the mouse pointer's coordinates.
  this._x = _root._xmouse;
  this._y = _root._ymouse;
  updateAfterEvent( );  // Refresh the screen for smooth movement.
}

this.onMouseDown = function ( ) {
  // When the mouse clicks down, shrink the clip.
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

```
  this._width  *= .5;
  this._height *= .5;
  updateAfterEvent( );
}

this.onMouseUp = function ( ) {
  // When the mouse clicks down, return the clip to its previous size.
  this._width  *= 2;
  this._height *= 2;
  updateAfterEvent( );
}
```

Example 10-4 creates a movie clip with button behaviors from ActionScript. It creates the clip on-the-fly with *MovieClip.createEmptyMovieClip( )*, then draws a square in the clip using the *MovieClip* drawing methods, and finally assigns the clip button handlers to create on, off, down, and up states. For more information on the drawing API, see Section 13.8 in Chapter 13.

Example 10-4. A movie clip button from scratch

```
// Create the clip
this.createEmptyMovieClip("optionButton_mc", 1);

// Draw a 16-pixel dark red square from (-8,-8) to (8,8)
optionButton_mc.lineStyle(undefined);
optionButton_mc.moveTo(-8,-8);
optionButton_mc.beginFill(0x990000);
optionButton_mc.lineTo(8, -8);
optionButton_mc.lineTo(8, 8);
optionButton_mc.lineTo(-8, 8);
optionButton_mc.lineTo(-8, -8);
optionButton_mc.endFill( );

// Turn the clip bright red when the mouse is over it
optionButton_mc.onRollOver = function ( ) {
  var c = new Color(this);
  c.setRGB(0xFF0000);
}

// Turn the clip bright red when the mouse is off of it
optionButton_mc.onRollOut = function ( ) {
  var c = new Color(this);
  c.setRGB(0x990000);
}

// Shrink the clip by 20% when it is pressed
optionButton_mc.onPress = function ( ) {
  this._xscale = this._yscale = 80;
}

// Restore the clip to its original size when it is released
optionButton_mc.onRelease = function ( ) {
  this._xscale = this._yscale = 100;
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

```
    }
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Chapter 10. Events and Event Handling

10.17 Onward!

With statements, operators, functions, and now events and event handlers under our belt, we've covered how all of the internal tools of ActionScript work. To round out your understanding of the language, in the next three chapters we'll explore three extremely important datatypes: arrays, objects, and movie clips.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Part I: ActionScript Fundamentals

Chapter 11. Arrays

Back in Chapter 3, we saw that primitive datatypes—strings, numbers, Booleans, `null`, and `undefined`—represent basic information in our scripts. We also saw that ActionScript supports several *composite* datatypes, which can group several pieces of data together into a single datum.

The *array* type is the first composite datatype we'll study. Arrays are used to store and manipulate ordered lists of information and are, therefore, fundamental tools in sequential, repetitive programming. We use them to do everything from storing values retrieved via a user-input form, to generating pull-down menus, to keeping track of enemy spacecraft in a game. In its most basic form, an array is just a list of items, like your grocery list or the entries in your checkbook ledger.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Chapter 11. Arrays

11.1 What Is an Array?

An array is a data structure that can encompass multiple individual data values, just like a building is a physical structure encompassing multiple floors. Unlike a primitive datatype, an array can include more than one data value. Here is a simple example showing two separate strings, followed by an array that contains two strings:

```
"oranges"                 // A single primitive string value
"apples"                  // Another primitive string value
["oranges", "apples"]     // A single array containing two strings
```

An array is a general-purpose container. It can contain any number of items, including items of different types. An array can even contain other arrays. Here is a simple example showing an array that contains both strings and numbers. It might represent your shopping list and how many of each item you intend to buy:

```
shoppingList = ["oranges", 6, "apples", 4, "bananas", 3];
```

Here's another analogy to make the concept of an array a little more tangible. Consider a chest of drawers. An individual drawer contains some content (socks, shirts, etc.). But the chest itself doesn't contain the *content*; it contains the *drawers*. The drawers hold the content. The chest organizes the drawers into a single unit.

When we work with an array (the chest of drawers), we usually are interested in the values within the array (the contents of the drawers). But we can also manipulate the containing structure itself. For example, we can change an array's size (add or subtract drawers) or reorder its values (swap the contents of its drawers).

Though we speak of an array as containing many values, it's important to recognize that the array itself is a single datum, in the same way that a string containing multiple characters is still a single string, and a number containing several digits is still a single number. As a single datum, an array can be assigned to a variable or used as part of a complex expression:

```
product = ["ladies downhill skis", 475];  // Store an array in a variable
display(product);                         // Pass that array to a function
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Chapter 11. Arrays

11.2 The Anatomy of an Array

Each item stored in an array is called an array *element*, and each element has a unique numeric position (*index*) by which we can refer to it.

11.2.1 Array Elements

Like a variable, each array element can store any legal datum. An entire array, then, is akin to a collection of sequentially named variables, but instead of each item having a different name, each item has an element number (the first element is number 0, not number 1).

Figure 11-1 shows, conceptually, the structure of an array that contains three elements. Element 0 stores the value "Erica", element 1 stores the value "Slavik", and element 2 stores the value "Gary".

Figure 11-1. A sample array structure

To manipulate the values in an array's elements, we ask for them by number. In our chest of drawers analogy, we might ask ActionScript to store something in the first drawer or retrieve whatever is in the second drawer.

11.2.2 Array Element Indexing

An element's position in the array is known as its *index*. Just as we can access the seventh character in a string, we can access the seventh element of an array via its index (in this case, the index is 6). We use an element's index to set or retrieve the element's value or to work with the element in various other ways. Some of the array-handling functions, for example, use element indexes to specify ranges of elements for processing.

We can also insert and delete elements from the beginning, end, or even middle of an array. An array can have gaps (that is, some elements can be absent). We can have elements at positions 0 and 4, without requiring anything in positions 1, 2, and 3. Arrays with gaps are called *sparse arrays*.

11.2.3 Array Size

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

At any point during its life span, an array contains a specific number of elements. The number of elements an array can hold is called the array's *length*, which we'll discuss later in this chapter.

Arrays in Other Programming Languages

Almost every high-level computer language supports arrays or array-like entities. That said, there are differences in the ways arrays are implemented across different languages. For example, many languages do not allow arrays to contain differing types of data. In many languages, an array can contain numbers or strings, but not both in the same array. Interestingly, in C, there is no primitive *string* datatype. Instead, C has a single-character datatype named *char*; strings are considered a complex datatype and are implemented as an array of *char*s.

In ActionScript, the size of an array changes automatically as items are added or removed. In many languages, the size of an array must be specified when the array is first *declared* or *dimensioned* (i.e., when memory is *allocated* to hold the array's data). Lingo, the scripting language for Macromedia Director, refers to its arrays by the name *lists*. Like ActionScript, Lingo allows arrays to contain data values of differing types, and it resizes its arrays automatically as needed. Unlike ActionScript and C, in which the first item in an array is numbered 0 (i.e., indexes are *zero-relative*), the first item in a Lingo list is numbered 1 (i.e., indexes are *one-relative*).

Languages differ as to what happens when you attempt to access an element outside the bounds (limits) of the array. ActionScript and Lingo add elements if you attempt to set a value for an element beyond the existing bounds of the array. If you attempt to access an element outside the array bounds, ActionScript returns `undefined`, whereas Lingo issues an error message. C pays no attention to whether you are accessing a valid element number. It lets you retrieve and set elements outside the bounds of the array, which usually causes you to overwrite other data in memory or access meaningless data that is not part of the array (C gives you plenty of rope with which to hang yourself). C even lets you specify a negative index, whereas ActionScript does not.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Chapter 11. Arrays

11.3 Creating Arrays

We can create a new array with a data literal (i.e., by simply specifying its elements in a comma-separated list enclosed in brackets) or with the built-in array *constructor* function, *Array( )* .

11.3.1 Array Literals

Usually, it's more convenient to create arrays using array literals than to create them with the *Array( )* constructor. Recall that *literal* is a direct representation of a piece of fixed data. For example:

```
"beaver"      // A string literal
234.2034      // A numeric literal
true          // A boolean literal
```

In an array literal, square brackets demarcate the beginning and end of the array. Inside the square brackets, the values of the array's elements are specified as a comma-separated list. Here's the general syntax:

```
[expression1, expression2, expression3]
```

The expressions are resolved and then stored as the elements of the array being defined. Any valid expression can be used, including function calls, variables, literals, and even other arrays (an array within an array is called a *nested* array or a *two-dimensional* array).

Here are a few examples:

```
[4, 5, 63];                                 // Simple numeric elements
["jeremy", "janice", "eman"]                // Simple string elements
[1, 4, 6 + 10]                              // Numeric expressions with an operat
[firstName, lastName, "tall", "skinny"]     // Variables and strings as elements
["month end days", [31, 30, 28]]            // A nested array literal
```

For comparison, let's do the same thing with the *Array( )* constructor:

```
new Array(4, 5, 63)
new Array("jeremy", "janice", "eman")
new Array(1, 4, 6 + 10)
new Array(firstName, lastName, "tall", "skinny")
new Array("month end days", new Array(31, 30, 28))
```

As shown in the preceding example and noted in the Arrays in Other Programming Languages sidebar, elements of a single ActionScript array can contain data of different types.

11.3.2 The Array Constructor

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

To create an array with the *Array( )* constructor, we use the *new* operator, followed by the word `Array` , followed by parentheses, which yields an empty array (one with no elements). As we'll see in Chapter 12, constructor functions create a ne data object, in this case an *Array* instance. We normally assign a newly created array to a variable or other data container for future reference. For example:

```
var myList = new Array();  // Store an empty array in variable myList
```

We often want to assign initial values to an array's elements. We can do so by passing parameters to the *Array( )* constructor v invoking it. Depending on the parameters we supply, the constructor invocation has different effects.

When we supply more than one argument to the *Array( )* constructor, or when we supply a single nonnumeric argument to the *Array( )* constructor, each argument becomes one of the element values in our new array. For example:

```
var frameLabels = new Array("intro", "section1", "section2", "home");
```

The array stored in `frameLabels` has the following elements:

```
0: "intro"
1: "section1"
2: "section2"
3: "home"
```

When we supply exactly one numeric argument to the *Array( )* constructor, it creates an array with the specified number of er placeholder elements:

```
var myList = new Array(14); // Creates an array with 14 empty elements
```

Arguments passed to the *Array( )* constructor can be any legal expression, including compound expressions:

```
var x = 10;
var y = 5;
var myNumbers = new Array(x + 1, x * y, Math.random( ));
```

The `myNumbers` variable stores an array with the following elements:

```
0: 11
1: 50
2: a floating-point number between 0 and 1
```

We mention the *Array( )* constructor here because it can create an array with placeholder elements, which is more cumberson achieve with an array literal. Furthermore, it serves as a gentle introduction to constructor functions. In practice, you'll usually create your arrays using array literals or by importing them from an external source. The most common use for the *Array( )* constructor is to create an empty array (possibly with a given number of placeholder elements).

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Chapter 11. Arrays

11.4 Referencing Array Elements

Once we've created an array, we'll inevitably want to retrieve or change the value of its elements. To do so, we can use the *array access operator* (i.e., square brackets), `[ ]`, which we introduced in Chapter 5.

11.4.1 Retrieving an Element's Value

To access an individual element, specify the array's identifier, followed by the element's index within square brackets, like this:

```
arrayName[elementNumber]
```

where *`arrayName`* is usually a variable pointing to the array, and *`elementNumber`* is an integer value specifying the element's index. The first element is number 0, and the last element number is 1 less than the array's length. Specifying an element number greater than the last valid element number causes the interpreter to return `undefined`. For example:

```
// Create an array using an array literal, and store it in trees
var trees = ["birch", "maple", "oak", "cedar"];

// Display the first element of trees in the Output window
trace(trees[0]);  // Displays: "birch"

// Assign the third element's value to the variable favoriteTree
// (remember indexes start at 0, so index 2 is the third element!)
var favoriteTree = trees[2];  // favoriteTree becomes "oak"
```

Now here's the fun part. Since we can specify the index of an element as any number-yielding expression, we can use variables or complex expressions just as easily as we use numbers to specify an element index. For example:

```
var i = 3;
var lastTree = trees[i];  // Set lastTree to "cedar"
```

We can even use function-call expressions that have numeric return values as our array indexes:

```
// Set randomTree to a randomly picked element of trees
// by calculating a random number between 0 and 3
var randomTree = trees[Math.floor(Math.random( ) * 4)];
```

Hot dog, that's powerful! You might use a similar approach to pick a random question from an array of trivia questions or to pick a random card from an array that represents a deck of cards.

Note that accessing an array element is very similar to accessing a variable value. Array elements can be used as part of any complex expression, as follows:

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

```
var myNums = [12, 4, 155, 90];
var myTotal = myNums[0] + myNums[1] + myNums[2] + myNums[3];   // Sum the arra
```

Summing the values of an array's elements manually isn't exactly the paragon of optimized code. Later, we'll see a much more convenient way to access an array's elements sequentially.

11.4.2 Setting an Element's Value

To set an element's value, we use *arrayName* [*elementNumber*] as the left-hand operand of an assignment expression:

```
// Make an array
var cities = ["Toronto", "Montreal", "Vancouver", "Waterloo"];
// cities is now: ["Toronto", "Montreal", "Vancouver", "Waterloo"]

// Set the value of the array's first element
// cities becomes ["London", "Montreal", "Vancouver", "Waterloo"]
cities[0] = "London";

// Set the value of the array's fourth element
// cities becomes ["London", "Montreal", "Vancouver", "Hamburg"]
cities[3] = "Hamburg";

// Set the value of the array's third element
// cities becomes ["London", "Montreal", 293.3, "Hamburg"]
cities[2] = 293.3;  // Notice that the datatype change is not a problem
```

Note that we can use any nonnegative numeric expression as the index when setting an array element:

```
var i = 1;
// Set the value of element i
// cities becomes ["London", "Prague", 293.3, "Hamburg"]
cities[i] = "Prague";
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Chapter 11. Arrays

11.5 Determining the Size of an Array

All arrays come with a built-in property named `length` , which indicates the current number of elements (including undefined elements). To access an array's `length` property, we use the dot operator, like so:

```
arrayName.length
```

Here are a few examples:

```
myList = [34, 45, 57];
trace(myList.length);              // Displays: 3
myWords = ["this", "that", "the other"];
trace(myWords.length);             // Displays: 3, the number of elements,
                                   // not the number of words or characters
frameLabels = new Array(24);       // Note the single numeric argument
                                   // used with the Array( ) constructor
trace(frameLabels.length);         // Displays: 24
```

The `length` of an array is always 1 greater than the index of its last element. For example, an array with elements at indexes 0, 1, and 2 has a length of 3. And an array with elements at indexes 0, 1, 2, and 50 has a length of 51. 51? Yes, 51. Even though indexes 3 through 49 are empty, they still contribute to the length of the array. The last element of an array is always `myArray [myArray .length - 1]` , because index numbers begin at 0, not 1.

If we add and remove elements, the array's `length` property is updated to reflect our changes. In fact, we can even set the `length` property to add or remove elements at the end of an array. This is in contrast the string.`length` property, which is read-only, as discussed in Chapter 4. Shortening the `length` of an array deletes elements beyond the new `length` . The `length` property includes numbered elements only, not named elements, as described in Section 11.6.

Using an array's `length` property, we can create a loop that accesses all the elements of an array, as we saw in Example 8-1. Looping through an array's elements is a fundamental task in programming. To get a sense of what's possible when we combine loops and arrays, study Example 11-1, which hunts through a `soundtracks` array to find the location of the element with the value "hip hop". You should recognize the *for* loop from Chapter 8, and the increment operator from Chapter 5. As for the array-access code, well, you just finished learning about that.

Example 11-1. Searching an array

```
// Create an array
var soundtracks = ["electronic", "hip hop", "pop", "alternative", "classical"]

// Check each element to see if it contains "hip hop"
for (var i = 0; i < soundtracks.length; i++) {
  trace("Now examining element: " + i);
  if (soundtracks[i] =  = "hip hop") {
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

```
    trace("The location of 'hip hop' is index: " + i);
    break;
  }
}
```

Let's extend Example 11-1 into a generalized search function that can check any array for any matching element, as shown in Example 11-2. Our search function returns the position within the array where the element was found, or `null` if it was not found.

Example 11-2. A generalized array-searching function

```
function searchArray (whichArray, searchElement) {

  // Check each element to see if it contains searchElement
  for (var i = 0; i < whichArray.length; i++) {
    if (whichArray[i] =  = searchElement) {
      return i;
    }
  }
  return null;
}
```

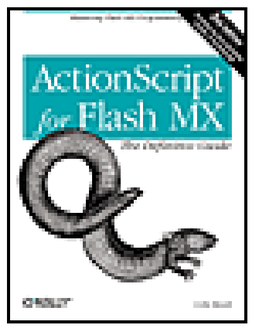
Here's how you might make use of our new search function to check whether or not "Fritz" is one of the names in our `userNames` array, which is an array of authorized usernames:

```
if (searchArray(userNames, "Fritz") =  = null) {
  trace("Sorry, that username wasn't found");
} else {
  trace("Welcome to the game.");
}
```

Now that's invigorating! This is one of those rewarding moments when all our hard work comes together in a single system. Seemingly trivial individual operations can combine to form extremely powerful and flexible programs. Like the letters in the alphabet or the sequence of base-pairs in a strand of DNA, you can construct anything imaginable from the simple building blocks at your disposal. The remainder of this chapter explains more about the mechanics of manipulating arrays, including the use of built-in functions that already perform some common functions for you. As you encounter needs not met by the built-in functions, consider writing your own custom functions, such as the *searchArray( )* function in the preceding example.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir



Chapter 11. Arrays

11.6 Named Array Elements

Numbered elements in an array are usually uniform in nature. For example, in an array of usernames, each element is a single user's name. But sometimes we'll want to store a bunch of related information in an array, for convenience, when the elements are not necessarily uniform. For example, in an array describing our computer, one element might be the processor speed, another might be the amount of RAM, and another might be the name of the operating system. The elements are related by virtue of describing our computer system, but the elements themselves are not uniform. Therefore, describing the elements as items 0, 1, and 2 may be accurate, but it isn't very descriptive. Although elements are usually numbered, fortunately, they can also be named. Named elements allow us to refer to elements in an array by a meaningful label—such as "processorSpeed", "ramInMB", or "osName"—instead of an integer. Not only does this make our code more legible, it also means we can access an element in an array without knowing its position. In Lingo, arrays that support named elements are called *property lists*. In many languages, they are known as *hashes*; in Perl, they are also called *associative arrays*.

Unlike in some languages, which allow an array element to be accessed by either name or number, a named ActionScript element cannot be accessed by number. Furthermore, named array elements cannot be manipulated by the Array methods (*push( )*, *pop( )*, etc., covered later in this chapter) and are not considered part of the array's numbered element list. For example, an array with one named element and two numbered elements will have a `length` of 2, not 3. To access all the named elements in an ActionScript array, therefore, we must use a *for-in* loop (discussed in Chapter 8), which lists both named and numbered elements.

> An array's named elements are simply object properties. In ActionScript, a generic object can hold a group of related properties as legitimately as an array can hold named elements. Both approaches are common practice.

11.6.1 Creating and Referencing Named Array Elements

To add an element that can later be retrieved by name, we use the familiar square brackets, with a string in place of the index number, on an existing array:

```
arrayName[elementName] = expression
```

where `elementName` is a string. For example:

```
var importantDates = new Array();                 // Create an empty array
importantDates["dadsBirthday"] = "June 1";         // Add a named element
importantDates["mumsBirthday"] = "January 16";    // Add another named element
```

Remember to create an empty array using the *new Array( )* constructor before trying to add named elements to it. We can also use the dot operator to specify a named array element, as follows:

```
arrayName.elementName = expression
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

In this case, *elementName* must be an identifier, *not* a string. For example:

```
var importantDates = new Array();
importantDates.dadsBirthday = "June 1";
importantDates.mumsBirthday = "January 16";
```

Assuming that we know an element's identifier (for example, `dadsBirthday` in the `importantDates` array), we can access it in one of two ways:

```
var goShopping = importantDates["dadsBirthday"];
var goShopping = importantDates.dadsBirthday;
```

Just as is the case when assigning a value to a named element, when accessing an element with square brackets, *elementName* must be a string or an expression that yields a string. Likewise, when used with the *dot* operator, *elementName* must be an *identifier* (i.e., the element's name without quotes), not a string. Note that when square brackets are used, *elementName* need not conform to the rules of forming a legal identifier. For example, the following *elementName* is not a legal identifier, but is permitted by the `[ ]` operator:

```
importantDates["dad's Birthday"] = "June 1";
```

The equivalent dot operator expression generates a compile-time error:

```
importantDates.dad's Birthday = "June 1";  // ERROR!
```

As a matter of good form, you should always avoid using illegal identifiers as named elements, despite the flexibility of the `[ ]` operator.

11.6.2 Removing Named Elements

To rid an array of an unwanted named element, we use the *delete* operator, which we introduced in Chapter 5:

```
delete arrayName.elementName
```

Deleting a named element destroys both the element value and the element container, freeing up any memory being occupied by the element and its contents. (Contrast this with the *delete* operator's behavior for *numbered* elements, where it simply clears the value but leaves the container intact.)

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Chapter 11. Arrays

11.7 Adding Elements to an Array

You can add elements to an array by specifying a value for a new element, increasing the array's `length` property, or using the built-in array functions.

11.7.1 Adding New Elements Directly

We can add a new element to an existing array at a specific index simply by assigning a value to that element:

```
// Create an array, and assign it three values
var myList = ["apples", "oranges", "pears"];

// Add a fourth value
myList[3] = "tangerines";
```

The new element does not need to be placed immediately after the last element of the old array. If we place the new element r than one element beyond the end of the array, ActionScript automatically creates undefined elements for the intervening inde

```
// Leave indexes 4 to 38 empty
myList[39] = "grapes";

trace(myList[12]); // Displays: undefined
```

If the element already exists, it will be replaced by the new value. If the element doesn't exist, it will be added.

11.7.2 Adding New Elements with the length Property

To extend an array without assigning values to new elements, we can simply increase the `length` property, and ActionScrip add enough elements to reach that length:

```
// Create an array with three elements
var myColors = ["green", "red", "blue"];
// Add 47 empty elements, numbered 3 through 49, to the array
myColors.length = 50;
```

You can use this approach to create a number of empty elements to hold some data you expect to accumulate, such as student scores.

11.7.3 Adding New Elements with Array Methods

We can use built-in array methods to handle more complex array operations. (In Chapter 12 you'll learn that a *method* is a fu that operates on an object.)

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

11.7.3.1 The push( ) method

The *push( )* method appends one or more elements to the end of an array. It automatically appends the data after the last numb element of the array, so we don't need to worry about how many elements already exist. The *push( )* method can also append r elements to an array at once. To invoke *push( )* on an array, we use the array name, followed by the dot operator, the keyworc and zero or more items in parentheses:

```
arrayName.push(item1, item2,...itemn);
```

where `item1` , `item2` , ...`itemn` is a comma-separated list of items to be appended to the end of the array as new elem Here are some examples:

```
// Create an array with two elements
var menuItems = ["home", "quit"];

// Add an element
// menuItems becomes ["home", "quit", "products"]
menuItems.push("products");

// Add two more elements
// menuItems becomes ["home", "quit", "products", "services", "contact"]
menuItems.push("services", "contact");
```

The *push( )* method returns the new length of the updated array (i.e., the value of the `length` property, which excludes nam elements):

```
var myList = [12, 23, 98];
trace(myList.push(28, 36));  // Appends 28 and 36 to myList and displays: 5
```

Note that the items added to a list can be any expression. The expression is resolved before being added to the list:

```
var temperature = 22;
var sky = "sunny";
var weatherListing = new Array();
weatherListing.push(temperature, sky);
trace(weatherListing); // Displays: "22,sunny", not "temperature,sky"
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Pushing, Popping, and Stacks

The *push( )* method takes its name from a programming concept called a *stack* . A stack can be thought of as a vertical array, like a stack of dishes. If you frequent cafeterias or restaurants with buffets, you should be familiar with the spring-loaded racks that hold plates for the diners. When clean dishes are added, they are literally *pushed* onto the top of the stack, and the older dishes sink lower into the rack. When a customer *pops* a dish from the top of the stack, he is removing the dish that was most recently pushed onto the stack. This is known as a last-in-first-out (LIFO) stack and is typically used for things like history lists. For example, if you hit the Back button in your browser, it will take you to the previous web page you visited. If you hit the Back button again, you'll be brought to the page before that, and so on. This is achieved by *pushing* the URL of each page you visit onto the stack and *popping* it off when the Back button is clicked.

LIFO stacks can also be found in real life. The last person to check her luggage on an airplane usually receives her luggage first when the plane lands, because the luggage is unloaded in the reverse order from which it was loaded. The early bird who checked his luggage first is doomed to wait the longest at the luggage conveyor belt after the plane lands. A first-in-first-out (FIFO) stack is more egalitarian—it works on a first-come-first-served basis. A FIFO stack is like the line at your local bank. Instead of taking the last element in an array, a FIFO stack deals with the first element in an array next. It then deletes the first element in the array, and all the other elements "move up," just as you move up in line when the person in front of you is "deleted" (i.e., either she is served and then leaves, or she chooses to leave in disgust because she is tired of waiting). Therefore, the word *push* generally implies that you are using a LIFO stack, whereas the word *append* implies that you are using a FIFO stack. In either case, elements are added to the "end" of the stack; the difference lies in which end of the array holds the element that is taken for the next operation.

11.7.3.2 The unshift( ) method

The *unshift( )* method is much like *push( )* , but it adds one or more elements to the *beginning* of the array, bumping all existi elements further along (i.e., the indexes of existing elements increase to accommodate the new elements at the beginning of tl The syntax of *unshift( )* follows the same style as all other array methods:

```
arrayName.unshift(item1, item2,...itemn);
```

where *item1* , *item2* , ...*itemn* is a comma-separated list of items to be added to the beginning of the array as new el Note that multiple items are added in the order that they are supplied. Here are some examples:

```
var flashVersions = new Array();
flashVersions[0] = 6;
flashVersions.unshift(5);      // flashVersions is now [5, 6]
flashVersions.unshift(2,3,4);  // flashVersions is now [2, 3, 4, 5, 6]
```

The *unshift( )* method, like *push( )* , returns the length of the newly enlarged array.

11.7.3.3 The splice( ) method

The *splice( )* method can add elements to, or remove elements from, an array. It is typically used to insert elements into the m an array (latter elements are renumbered to make room) or to delete elements from the middle of an array (latter elements are renumbered to close the gap). When *splice( )* performs both of these tasks in a single invocation, it effectively replaces some with new elements (though not necessarily with the same number of elements). Here's the syntax for *splice( )* :

```
arrayName.splice(startIndex, deleteCount, item1, item2,...itemn)
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

where *startIndex* is a number that specifies the index at which element removal and optional insertion should commen (remember that the first element's index is 0); *deleteCount* is an optional argument that dictates how many elements sh removed (including the element at *startIndex*). When *deleteCount* is omitted, every element after and including *startIndex* is removed. The optional *item1* , *item2* , ...*itemn* parameters are items to be added to the array as e starting at *startIndex* .

Example 11-3 shows the versatility of the *splice( )* method.

Example 11-3. Using the splice( ) array method

```
// Make an array...
months = new Array("January", "Friday", "April", "May", "Sunday", "Monday", "J
// Hmmm. Something's wrong with our array. Let's fix it up.
// First, let's get rid of "Friday".
months.splice(1,1);
  // months is now:
  // ["January", "April", "May", "Sunday", "Monday", "July"]

// Now, let's add the two months before "April".
// Note that we won't delete anything here (deleteCount is 0).
months.splice(1, 0, "February", "March");
  // months is now:
  // ["January", "February", "March", "April", "May", "Sunday", "Monday", "Jul

// Finally, let's remove "Sunday" and "Monday" while inserting "June".
months.splice(5, 2, "june");
  // months is now:
  // ["January", "February", "March", "April", "May", "June", "July"]

// Now that our months array is fixed, let's trim it
// so that it contains only the first quarter of the year,
// by deleting all elements starting with index 3 (i.e., "April").
months.splice(3); // months is now: ["January", "February", "March"]
```

Another useful feature of *splice( )* is that it returns an array of the elements it *removes* . Thus it can be used to extract a series elements from an array:

```
myList = ["a", "b", "c", "d"];
trace(myList.splice(1, 2));  // Displays: "b,c"
                             // myList is now ["a", "d"]
```

If no elements are removed, *splice( )* returns an empty array (that is, an array with no elements).

11.7.3.4 The concat( ) method

Like *push( )* , *concat( )* adds elements to the end of an array. Unlike *push( )* , *concat( )* does not modify the array on which it i invoked—instead, *concat( )* returns a new array. Furthermore, *concat( )* can combine two or more arrays into a single, new arr Here's the syntax for *concat( )* :

```
origArray.concat(elementList)
```

The *concat( )* method appends the elements contained in *elementList* , one by one, to the end of *origArray* and re

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

result as a *new* array, leaving `origArray` untouched. Normally, we store the returned array in a variable. Here, simple nu are used as the items to be added to the array:

```
var list1 = new Array(11, 12, 13);
var list2 = list1.concat(14, 15);  // list2 becomes [11, 12, 13, 14, 15]
```

In the following example, we use *concat( )* to combine two arrays:

```
var guests = ["Panda", "Dave"];
var registeredPlayers = ["Gray", "Doomtrooper", "TRK9"];
var allUsers = registeredPlayers.concat(guests);
// allUsers is now: ["Gray", "Doomtrooper", "TRK9", "Panda", "Dave"]
```

Notice that *concat( )* separated the elements of the `guests` array in a way that *push( )* would not have. If we had tried this c

```
var allUsers = registeredPlayers.push(guests);
```

we'd have ended up with this nested array:

```
["Gray", "Doomtrooper", "TRK9", ["Panda", "Dave"]]
```

Furthermore, *push( )* would have altered the `registeredPlayers` array, whereas *concat( )* does not.

Note, however, that *concat( )* cannot break apart *nested* arrays (elements that are themselves arrays within the main array), as see from the following code:

```
var x = [1, 2, 3];
var y = [[5, 6], [7, 8]];
var z = x.concat(y);  // Result is [1, 2, 3, [5, 6], [7, 8]].
                      // Elements 0 and 1 of y were not "flattened."
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Chapter 11. Arrays

11.8 Removing Elements from an Array

You can remove elements from an array by using the *delete* operator, reducing the `length` property of an array, or using one of the built-in array methods.

11.8.1 Removing Numbered Elements with the delete Operator

The *delete* operator sets an array element to `undefined` , using the following syntax:

```
delete arrayName[index]
```

where *`arrayName`* is any array and *`index`* is the number or name of the element we want to set to `undefined` . When used with numbered (not named) elements, the name *delete* is misleading, frankly. It does *not* remove a numbered element from the array; it merely sets the target element's value to `undefined` . A *delete* operation, therefore, is identical to assigning the `undefined` value to an element. We can verify this by checking the `length` property of an array after deleting one of its elements:

```
var myList = ["a", "b", "c"];
trace(myList.length);  // Displays: 3
delete myList[2];
trace(myList.length);  // Still displays 3...the element at index 2 is
                       // undefined instead of "c", but it still exists
```

To truly delete elements, use *splice( )* (to delete them from the middle of an array), or use *shift( )* and *pop( )* (to delete them from the beginning or end of an array). Note that *delete* behaves differently with named elements than with numbered elements. Using *delete* on named elements destroys them permanently, leaving no trace. As we saw in Chapter 5, *delete* can also destroy object properties as it does named array elements (in fact, named elements are actually object properties of the *Array* object).

11.8.2 Removing Elements with the length Property

Earlier we used the `length` property to add elements to an array. We can also set the array's `length` property to a number smaller than the current `length` in order to delete elements from the array (i.e., truncate the array):

```
var toppings = ["pepperoni", "tomatoes", "cheese", "green pepper", "broccoli"]
toppings.length = 3;
trace(toppings);  // Displays: "pepperoni,tomatoes,cheese"
                  // We trimmed elements 3 and 4 (the last two).
```

11.8.3 Removing Elements with Array Methods

Arrays come equipped with several built-in methods for removing elements. We've already seen how *splice( )* can delete a series of elements from the middle of an array. The *pop( )* and *shift( )* methods are used to prune elements from the end or

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

beginning of an array.

11.8.3.1 The pop( ) method

The *pop( )* method is the antithesis of *push( )* —it removes the last element of an array. The syntax of *pop( )* is simple:

```
arrayName.pop()
```

I don't know why, but I always think that "popping" an array is kinda funny. Anyway, *pop( )* decrements the array's `length` by 1 and returns the value of the element it removed. For example:

```
x = [56, 57, 58];
trace(x.pop());  // Displays: 58 (the value of the popped element)
                 // x is now [56, 57]
```

As we saw earlier, *pop( )* is often used in combination with *push( )* to perform LIFO stack operations. In Example 11-4, we create a `siteHistory` array that tracks a user's navigation through a site. When the user navigates to a new frame, we add his location to the array using *push( )* . When the user navigates back, we *pop( )* his last location off of `siteHistory` and send him to the preceding location. Example 11-4 can be downloaded from the online Code Depot.

Example 11-4. A Back button with history

```
// CODE ON FRAME 1 OF OUR MOVIE
stop( );
var siteHistory = new Array( );

function goto (theLabel) {
  // If we're not already at the requested frame...
  if (theLabel != siteHistory[siteHistory.length - 1]) {
    // ...add the request to the history, then go to the requested frame
    siteHistory.push(theLabel);
    this.gotoAndStop(siteHistory[siteHistory.length - 1]);
  }
  trace(siteHistory);
}

function goBack () {
  // Remove the last item in the history
  siteHistory.pop();
  // If there is anything left in the history...
  if (siteHistory.length > 0) {
    // ...go to the most recent frame
    this.gotoAndStop(siteHistory[siteHistory.length - 1]);
  } else {
    // ...otherwise go home
    this.gotoAndStop("home");
  }
  trace(siteHistory);
}

// NAVIGATION BUTTON
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

```
gallery_btn.onRelease = function ( ) {
  this._parent.goto("gallery");
}

// THE BACK BUTTON
back_btn.onRelease = function ( ) {
  this._parent.goBack();
}
```

11.8.3.2 The shift( ) method

Remember *unshift( )* , the method we used to add an element to the beginning of an array? Meet its alter ego, *shift( )* , which removes an element from the beginning of an array:

```
arrayName.shift()
```

Not as funny as *pop* . Oh well.

Like *pop( )* , *shift( )* returns the value of the element it removes. The remaining elements all move up in the pecking order toward the beginning of the array. For example:

```
var sports = ["hackey sack", "snowboarding", "inline skating"];
trace(sports.shift( ));  // Displays: hackey sack
                         // sports is now ["snowboarding", "inline skating"]
trace(sports.shift( ));  // Displays: snowboarding
                         // sports is now ["inline skating"]
```

Because *shift( )* truly deletes an element, it is more useful than *delete* for removing the first element of an array. We can also use *shift( )* to limit the number of elements in a list. For example, suppose we're calculating the frame rate of a movie. We *push( )* the current time onto an array after each frame renders. To limit the size of our array to the most recent 10 time samples, we *shift( )* the oldest time off as necessary. To find the frame rate, we average the times in our array. Example 11-5 shows this technique.

Example 11-5. Calculating the frame rate of a movie

```
// Create a movie clip to track the frame rate
this.createEmptyMovieClip("fpsTracker", 1);

// Create our time measurement array in the fpsTracker movie clip
fpsTracker.elapsedTime = new Array();

// Use an onEnterFrame( ) event handler to measure the time after each frame
fpsTracker.onEnterFrame = function ( ) {
  // Add the current time to elapsedTime
  this.elapsedTime.push(getTimer( ));

  // If we have enough samples to calculate an average...
  if (this.elapsedTime.length > 10) {
    // ...remove the oldest time from elapsedTime
    this.elapsedTime.shift();

    // Average the number of elapsed milliseconds per frame
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

```
        this.elapsedAverage = (this.elapsedTime[this.elapsedTime.length - 1] -
                               this.elapsedTime[0]) / this.elapsedTime.length;

        // To find the frames per second, divide 1 second by the elapsed average
        this.fps = 1000 / this.elapsedAverage;
        trace("Current fps " + this.fps);
    }
}
```

11.8.3.3 The splice( ) method

Earlier we saw that *splice( )* can both remove elements from and add elements to an array. Since we've already looked at *splice( )* in detail, we won't reexamine it here. However, given our current context, we should specifically demonstrate *splice( )* 's element-removal capabilities:

```
var x = ["a", "b", "c", "d", "e", "f"];
x.splice(1,3);  // Removes elements 1, 2, and 3, leaving ["a", "e", "f"]
x.splice(1);    // Removes elements 1 through the end leaving just ["a"]
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Chapter 11. Arrays

11.9 General Array-Manipulation Tools

We've now seen how array methods can be used to remove elements from and add elements to arrays. ActionScript also offer built-in methods for reordering and sorting elements, converting array elements to strings, and extracting arrays from other arrays.

11.9.1 The reverse( ) Method

As its name suggests, the *reverse( )* method reverses the order of the elements of an array. This method is simple but impressi Here's the syntax:

```
arrayName.reverse()
```

And here are the results:

```
var x = [1, 2, 3, 4];
x.reverse();
trace(x);  // Displays: "4,3,2,1"
```

We typically use *reverse( )* to reorder a sorted list. For example, if we have a list of products sorted by ascending price, we ca display them from least to most expensive, or we can reverse the list to display them from most to least expensive.

Reader Exercise: Try to write your own custom function to reverse the elements in an array. Not only is it harder than it looks you'll most likely find that the built-in *reverse( )* method is substantially faster.

11.9.2 The sort( ) and sortOn( ) Methods

The *sort( )* method rearranges elements in an array according to an arbitrary rule that we provide. If we provide no rule, *sort( )* places the elements in (roughly) alphabetical order by default. Sorting an array alphabetically is really easy, so let's first see h that works:

```
arrayName.sort()
```

When we invoke an array's *sort( )* method with no arguments, its elements are temporarily converted to strings and sorted according to their Unicode code points (equivalent to ASCII values for code points below 128). For the code points of most western European languages, see Appendix B. See also Section 4.6.2.2 in Chapter 4 for important details.

```
// This works as expected...
var animals = ["zebra", "ape"];
animals.sort();
trace(animals);  // Displays: "ape,zebra"
                 // Cool! What a handy little method.
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

```
// Watch out, the sort order is not strictly alphabetical...
// The capital "Z" in zebra comes before the lowercase "a" in "ape"
var animals = ["Zebra", "ape"];
animals.sort();
trace(animals);  // Displays: "Zebra,ape". Oops. See Appendix B.
```

We can also use *sort( )* to organize array elements according to a rule of our own choosing. This technique is a little trickier t work with, but it's quite powerful. We start by creating a *compare function* that dictates how the interpreter should sort any tw elements of an array. We then pass that function to the *sort( )* method when we call it, like this:

```
arrayName.sort(compareFunction)
```

where *compareFunction* is the name of the function that tells the interpreter how to make its sorting decisions.

To build a compare function, we start with a new function that accepts two arguments (these represent any two elements in th array). In the function's body, we determine, however we see fit, which of the elements we'd like to appear earlier in the elem list after a sort operation. If we want the first element to appear *before* the second element, we return a *negative* number from function. If we want the first element to appear *after* the second element, we return a *positive* number from our function. If w want the elements to be left in the same positions, we return 0 from our function. In pseudocode, the approach generically loo like this:

```
function compareElements (element1, element2) {
  if (element1 should appear before element2) {
    return -1;
  } else if (element1 should appear after element2) {
    return 1;
  } else {
    return 0;  // The elements should be left alone
  }
}
```

For example, to put elements in ascending numeric order, we can use a function like this:

```
function sortAscendingNumbers (element1, element2) {
  if (element1 < element2) {
    return -1;
  } else if (element1 > element2) {
    return 1;
  } else {
    return 0;  // The elements are equal
  }
}

// Now that our compare function is ready, let's try it out
var x = [34, 55, 33, 1, 100];
x.sort(sortAscendingNumbers);
trace(x);    // Displays: "1,33,34,55,100"
```

Numeric sorting functions can actually be phrased much more succinctly. The preceding *sortAscendingNumbers( )* function c also be written as:

```
function sortAscendingNumbers (element1, element2) {
  return element1 - element2;
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

```
}
```

In our optimized version, a negative number is returned if `element1` is less than `element2` , a positive number is returne `element1` is greater than `element2` , and a 0 is returned if the two are equal. Now that's elegant! Here is a version to perform a descending sort:

```
function sortDescendingNumbers (element1, element2) {
  return element2 - element1;
}
```

Example 11-6 shows a compare function that adjusts the default alphabetic comparison behavior of *sort( )* so that upper- and lowercase letters are sorted without regard to case.

Example 11-6. Case-insensitive alphabetic array sort

```
var animals = ["Zebra", "ape"];

function sortAscendingAlpha (element1, element2) {
 return (element2.toLowerCase( ) < element1.toLowerCase());
}
animals.sort(sortAscendingAlpha);

trace(animals);  // Displays: "ape,Zebra"
```

Of course, the comparison does not always have to involve simple strings and numbers. Here we sort an array of movie clips ascending order, according to their pixel area:

```
var clips = [square1, square2, square3];

function sortByClipArea (clip1, clip2) {
  clip1area = clip1._width * clip1._height;
  clip2area = clip2._width * clip2._height
  return clip1area - clip2area;
}

clips.sort(sortByClipArea);
```

That's a mighty fine sortin' machine!

Flash MX adds another useful sorting method not available in Flash 5. When the elements of an array are objects that share a specific property name, we can use *sortOn( )* to sort the objects quickly according to that property. For details, see *Array.sort( )* in the ActionScript Language Reference.

11.9.3 The slice( ) Method

Something of a subset of *splice( )* , the *slice( )* method retrieves a series of elements from an array. Unlike *splice( )* , *slice( )* o *retrieves* elements. It creates a new array and does not affect the array on which it is invoked. The *slice( )* method has the following syntax:

```
origArray.slice(startIndex, endIndex)
```

where `startIndex` specifies the first element to retrieve and `endIndex` specifies the element *after* the last element w

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

want to retrieve. The *slice( )* method returns a new array containing a copy of the series of elements from `origArray [ startIndex ]` to `origArray [ endIndex - 1]` . If `endIndex` is omitted, it defaults to `origArray .length` , and the returned array contains the elements from `origArray [ startIndex ]` through `origArray origArray .length - 1]` . Here are a couple of examples:

```
var myList = ["a", "b", "c", "d", "e", "f"];
myList.slice(1, 3);  // Returns ["b", "c"], not ["b", "c", "d"]
myList.slice(2);     // Returns ["c", "d", "e", "f"]
```

11.9.4 The join( ) Method

We can use *join( )* to produce a string that represents all the numbered elements of an array. The *join( )* method starts by converting each element of the specified array to a string. Undefined elements are converted to the empty string (`""`). Then *j)* concatenates all the strings into one long string, separating them with a character (or series of characters) called a *delimiter* . Finally, *join( )* returns the resulting string. The syntax of *join( )* is:

```
arrayName.join(delimiter)
```

where `delimiter` is the string used to separate the converted elements of *arrayName* . If `delimiter` is unspecified, it defaults to a comma. The result of a *join( )* statement is most easily understood through an example:

```
var siteSections = ["animation", "short films", "games"];
// Sets siteTitle to "animation>> short films>> games"
var siteTitle = siteSections.join(">> ");
// Sets siteTitle to "animation:short films:games"
var siteTitle = siteSections.join(":");
```

Note that *join( )* does not modify the array upon which it is invoked. Instead, it returns a string based on that array.

When called without a delimiter argument, *join( )* behaves exactly like *toString( )* . Because *toString( )* does not add a space a the comma it uses to delimit elements, we may wish to use *join( )* with ", " as the delimiter if we want nicely formatted output

```
var x = [1, 2, 3];
trace(x.join(", "));  // Displays: "1, 2, 3" instead of "1,2,3"
```

The *join( )* method has a potentially surprising result when used on an array containing elements that are *themselves* arrays. Since *join( )* converts elements to strings, and arrays are converted to strings via their *toString( )* method, nested arrays are converted to strings using a comma delimiter, not the delimiter supplied as an argument to *join( )* . In other words, any delimi supplied to *join( )* doesn't affect nested arrays. For example:

```
var x = [1, [2, 3, 4], 5];
x.join("|");  // Returns "1|2,3,4|5" not "1|2|3|4|5"
```

Reader Exercise: Write your own custom function that uses the specified delimiter to join nested arrays. Hint: Loop through a the elements of the array, checking for those that are arrays themselves. Use the *instanceof* operator (described in Chapter 5) t check whether a datum is an array. Can you create a recursive version that will work with arrays nested to an arbitrary depth?

11.9.5 The toString( ) Method

As we'll see in Chapter 12, *toString( )* is a method, common to all objects, that returns a string representation of the object up which it is invoked. In the case of an *Array* object, the *toString( )* method returns a list of the array's elements, converted to strings and separated by commas. The *toString( )* method can be called explicitly:

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

```
arrayName.toString()
```

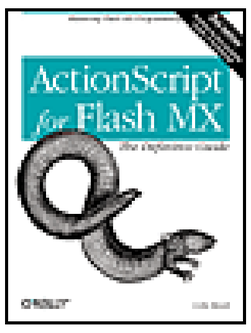
Typically, we don't use *toString( )* explicitly; rather, it is invoked automatically whenever `arrayName` is used in a string context. For example, when we write *trace(* `arrayName` *)* , a list of comma-separated values appears in the Output windov *trace(* `arrayName` *)* is equivalent to *trace(* `arrayName` *.toString( ))* . The *toString( )* method is often helpful during debugging when we need a quick, unformatted look at the elements of an array. For example:

```
var sites = ["www.moock.org", "www.macromedia.com", "www.oreilly.com"];
// Display our array in a text field
debugOutput_txt.text = "The sites array is " + sites.toString();

trace("The sites array is " + sites.toString());   // Explicit use of toString
trace("The sites array is " + sites);              // Implicit use of toString
```

The *join( )* method covered in the previous section offers greater formatting flexibility than *toString( )* .

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir



ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Chapter 11. Arrays

11.10 Arrays as Objects

Once you've read Chapter 12, you should recognize that arrays are a type of object, specifically, instances of the *Array* class. As with other objects, *Array* objects can have properties added to them. When we add named elements to an array, we're really just adding properties to an *Array* object. However, as we saw earlier, to use the built-in methods of the *Array* class we must work with numbered elements.

In Chapter 9, we saw a useful "array-object" hybrid, the *arguments* object, that combined numbered elements with properties (Recall that the *arguments* object has `callee` and `caller` properties but also stores an array of parameters as array elements.) Now we can see that all arrays contain their numbered list of elements, plus the built-in property `length` , plus any custom properties we add.

In addition, because arrays are instances of the *Array* class, we can add new *Array* methods that work on all arrays. For example, the following code creates a method named *searchArray( )* , based on the function we created earlier in Example 11-2.

```
// Add a new searchArray( ) method to all arrays
Array.prototype.searchArray = function (searchElement) {
  // Check each element to see if it contains searchElement
  for (var i = 0; i < this.length; i++) {
    if (this[i] =  = searchElement) {
      return i;
    }
  }
  return null;
};

// Create an array
var soundtracks = ["electronic", "hip hop", "pop", "alternative", "classical"]

// Use the searchArray method to find the index of the word "classical"
trace(soundtracks.searchArray("classical"));  // Displays: 4
```

To determine whether an object is an array, use the *instanceof* operator. For example:

```
var colors = ["red", "blue", "green"];
trace(colors instanceof Array);  // Displays: true
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Chapter 11. Arrays

11.11 Multidimensional Arrays

So far, we've limited our discussion to *one-dimensional* arrays, which are akin to a single row or a single column in a spreadsheet. But what if we want to create the equivalent of a spreadsheet with both rows and columns? We need a second dimension. ActionScript natively supports only one-dimensional arrays, but we can simulate a multidimensional array by creating arrays within arrays. That is, we can create an array that contains elements that are themselves arrays (sometimes called *nested* arrays).

The simplest type of multidimensional array is a two-dimensional array, in which elements are organized conceptually into a grid of rows and columns—the rows are the first dimension of the array, and the columns are the second.

Using a practical example, let's consider how a two-dimensional array works. Suppose we're processing an order that contains three products, each with a quantity and a price. We want to simulate a spreadsheet with three rows (one for each product) and two columns (one for the quantity and one for the price). We create a separate array for each row, with each row's elements representing the values in each column:

```
var row1 = [6, 2.99];   // Quantity 6, Price 2.99
var row2 = [4, 9.99];   // Quantity 4, Price 9.99
var row3 = [1, 59.99];  // Quantity 1, Price 59.99
```

Next, we place the rows into a container array named `spreadsheet`:

```
var spreadsheet = [row1, row2, row3];
```

Now we can find the total cost of the order by multiplying the quantity and price of each row and adding them all together. We access a two-dimensional array's elements using two indexes (one for the row and one for the column). The expression `spreadsheet[0]`, for example, represents the first row's two-column array. Hence, to access the second column in the first row of `spreadsheet`, we use `spreadsheet[0][1]`:

```
// Create a variable to store the total cost of the order.
var total;

// Now find the cost of the order. For each row, multiply the columns
// together, and add that to the total.
for (var i = 0; i < spreadsheet.length; i++) {
  total += spreadsheet[i][0] * spreadsheet[i][1];
}

trace(total);  // Displays: 117.89
```

Aside from storage and access, multidimensional arrays behave just like normal arrays. We can use all of the *Array* methods to manipulate data stored in multidimensional arrays.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Chapter 11. Arrays

11.12 The Multiple-Choice Quiz, Take 3

In Example 9-13, we revised the multiple-choice quiz from Chapter 1 (Examples Example 1-1 through Example 1-4). At that time, we improved the quiz code by creating reusable, centralized functions. Now that we've studied arrays, we can return to the quiz to make a few more improvements. This time, we'll use arrays to store both the user's answers and the correct answers for each question, making our code more succinct, legible, and easier to extend if we want to add new questions.

The changes we need to make to the quiz take place on the *scripts* layer of frame 1, where our main quiz code is located. Both the previous version and this updated version of the quiz can be found in the online Code Depot. To see what improvements we'll make this time around, read through the code in Example 11-7, paying special attention to the comments.

Example 11-7. A multiple-choice quiz, version 3

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

// =  =  =  =  =  =  =  =  =  =  =  =  =  =  =  =  =  =  =
// INIT QUIZ VARIABLES
// =  =  =  =  =  =  =  =  =  =  =  =  =  =  =  =  =  =  =
// Store a reference to the timeline that contains the quiz user interface.
var quizTimeline = this;
// Create an array to contain user's answers.
var userAnswers = new Array( );
// Create an array to contain each question's correct answer.
var correctAnswers = [3, 2];
// Convenience variable to store number of questions in quiz.
var numQuestions = correctAnswers.length;

// =  =  =  =  =  =  =  =  =  =  =  =  =  =  =  =  =  =  =  =  =
// CREATE QUIZ FUNCTIONS
// =  =  =  =  =  =  =  =  =  =  =  =  =  =  =  =  =  =  =  =  =
//   Function: answer( )
//       Desc: Registers the user's answers to quiz questions and
//             advances the quiz through its questions.
//     Params: choice           The user's answer for the current question.
function answer (choice) {
  // Add the current answer to the list of user answers.
  userAnswers.push(choice);

  // Convenience variable to store number of current question.
  var currentQuestion = userAnswers.length;
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

```
    // Display the question and answer in the Output window for debugging.
    trace("Question " + currentQuestion
            + ". The user answered: " + userAnswers[currentQuestion-1] + ".");

    // If we're on the last question...
    if (currentQuestion =  = numQuestions) {
      // ...go to the quiz end frame.
      quizTimeline.gotoAndStop("quizEnd");
      // And grade the user.
      gradeUser( );
    } else {
      // ...otherwise, go to the next question frame.
      quizTimeline.gotoAndStop("q"+ (currentQuestion + 1));
    }
}

//   Function: gradeUser( )
//       Desc: Tallies the user's score at the end of the quiz.
function gradeUser ( ) {
  // Report that we're about to grade the quiz in the Output window.
  trace("Quiz complete. Now grading...");

  // Create a local variable to track the
  // number of questions the user answered correctly.
  var totalCorrect = 0;

  // Count how many questions the user answered correctly.
  // For each question...
  for (var i=0; i < numQuestions; i++) {
    // If the user's answer matches the correct answer.
    if (userAnswers[i] =  = correctAnswers[i]) {
      // Give the user a point.
      totalCorrect++;
    }
    // Display the correct answer and the user's answer
    // in the Output window for debugging.
    trace("Question " + (i + 1)
          + ". Correct answer: " + correctAnswers[i]
          + ". User's answer: "  + userAnswers[i]);
}

// Display the final score in the Output window for debugging.
trace("User's score: " + totalCorrect + "/" + numQuestions);

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

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

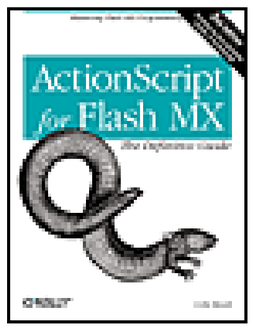
downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

```
  // Customize the font face, size, and color of the text field.
  var formatObj = new TextFormat( );
  formatObj.size = 16;
  formatObj.color = 0xC7FF9C;
  formatObj.font = "_sans";
  formatObj.bold = true;
  quizTimeline.totalOutput_txt.setTextFormat(formatObj);
}

// =  =  =  =  =  =  =  =  =  =  =  =  =  =  =  =  =  =  =  =  =  =  =
// DEFINE QUIZ BUTTON EVENT HANDLERS
// =  =  =  =  =  =  =  =  =  =  =  =  =  =  =  =  =  =  =  =  =  =  =
// Code executed when button 1 is pressed.
choice1_btn.onRelease = function ( ) {
  // Call answer( ), which records the user's choice
  // and advances the quiz to the next question.
  this._parent.answer(1);
};

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

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

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir



Chapter 11. Arrays

11.13 Onward!

This chapter gave you an introduction to—and considerable practice using—your first composite datatype, arrays. Now that you understand how multiple pieces of data can be represented within a single container, you have a solid foundation to take into Chapter 12. There we'll explore an even more powerful type of composite data—and a fundamental component of ActionScript—*objects*.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Part I: ActionScript Fundamentals

Chapter 12. Objects and Classes

This chapter covers *object-oriented programming (OOP)*, which is new territory for many readers. We'll cover OOP methodology and terminology, using applied examples to make it all concrete. For the benefit of both experienced programmers and new programmers who appreciate a broad perspective, we'll compare ActionScript's OOP practices with those of Java and other languages. Experienced object-oriented programmers who simply want an overview of ActionScript's OOP syntax may want to skip directly to the Section 12.9 at the end of this chapter.

If you're new to object-oriented programming, you may have heard that OOP is some big mystery or that it's difficult to understand. Quite to the contrary, the concepts are highly intuitive, and the OOP process is much easier than you may have been led to believe. At its heart, OOP simply means that you treat portions of your program as self-contained modules called "objects." By "self-contained," we don't mean that objects are isolated, but rather that they can be independent of each other, and that you can use an object without worrying about its internal details. OOP concepts are easy to grasp once you realize that everything you deal with in the real world is a self-contained object. Your dog, your parents, your car, and your computer are all self-contained objects—meaning that they do some things independently and do other things at your request, even if you don't know the inner details of how they work.

You don't have to be a biologist to get your dog to fetch a stick; you don't need to be a mechanical engineer to drive your car; you don't need to be a psychoanalyst to interact with your parents; and, rumors to the contrary, you don't need to be a computer scientist to check your email. All you need to know is the commands an object is willing to obey (which are called *methods*) and the results those commands produce. For example, if you press the gas pedal of your car, you expect the car to accelerate. If you tell your dog to sit, you expect him to sit. Armed with this commonsense context of what an object is in the real world, let's see how to relate these concepts to ActionScript.

The classic example of a programming *object* is a bouncing ball. Like a real ball, a ball object can have *properties* that represent its attributes, such as its radius, color, mass, position, and bounciness (elasticity). To represent our bouncing ball in a program, we'll create a `ball` object with a `radius` property, a `color` property, and so forth. The properties of an object represent its state at any given time, but some of its properties change over time. For example, to make our ball move, we need to simulate Newton's laws of motion. That is, we need to describe in computer terms how a ball moves over time. In the simplest case, recalling that speed multiplied by time equals distance, we can determine a ball's horizontal position in the future using this equation:

```
Position = Speed * Time
```

We can translate this equation into ActionScript that calculates the ball's new position:

```
ball.xPosition += ball.xVelocity * elapsedTime;
```

This equation starts with the ball's current position and adds the distance it has traveled (based on its velocity and the elapsed time) to come up with the new position.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

An object's *behaviors* are simply the things it can do or the services it can perform (one of our ball's behaviors is its ability to move). To describe an object's behaviors in code, we create so-called *methods*, which are simply the functions that implement an object's behaviors.

For example, we might create a *move( )* method on our `ball` object that uses the preceding equation. Therefore, methods can be thought of as the commands that an object obeys. Of course, an object can have multiple methods. Let's say we want to make our ball bounce. We can create a *bounce( )* method that reverses the ball's direction and reduces its speed (our ball isn't perfectly elastic). The *bounce( )* method might implement this equation:

```
ball.xVelocity =  -(ball.xVelocity) * 0.95   // Ball is 95% elastic
```

Before getting into the esoterica of how to create objects, add properties, and implement methods, let's formalize some of our definitions. An *object* is a data structure that groups together related *properties* (variables) and *methods* (functions). An object typically *encapsulates* its behaviors, meaning that the internal details of how it performs its functions are not necessarily visible outside the object. Instead, a program interacts with an object via its so-called *interfaces* (i.e., methods that are publicly accessible outside the object). The rest of the program doesn't have to worry about how an object does what it does; instead, the program provides *inputs* to the object and checks the *outputs* (results) when applicable.

You'll also hear talk of *classes* and *instances*. A *class* is a generic object category, and an *instance* is a specific case (i.e., a copy) of the class of object. For example, your particular pet dog is an instance of the generic *Dog* class. All dogs in the *Dog* class bark and have four legs, but your specific dog has its own particular values for the `height`, `weight`, and `color` properties used to describe dogs. If you've seen a three-legged dog and therefore just objected to the assertion that "all dogs have four legs," you're already understanding OOP—a class should accommodate all possible variations of its instances. Hence, to satisfy your objection to the *Dog* class design, you could add a `numberOfLegs` property that would be set separately for each individual *Dog* object.

Object-oriented programming (OOP) is merely the name given to programs that make use of objects. Objects and OOP are so intrinsic to ActionScript that we've already used them, perhaps without your realizing it. A movie clip is a familiar object in Flash, and like all objects, it is implemented as a collection of properties and methods. When we determine the height of a movie clip using `someClip._height`, we're accessing that clip object's `_height` property. And when we tell a movie clip to play using `someClip`*.play( )*, we're invoking that clip object's *play( )* method.

> Typically, all instances of an object class share the same methods and property names; it is the property *values* of each instance that distinguish it from other instances of the same class.

Whether we create objects ourselves or use those built into ActionScript, OOP keeps different parts of a program cleanly separated from one another (*encapsulated*) and allows them to interoperate without knowing the internal details of other objects. This allows an object to change its internal functionality without adversely affecting other portions of the program that rely on the object, so long as the object's public methods (i.e., its interfaces to the outside world) don't change. Returning to our `ball` object, for example, we don't care if the laws of physics change the behavior of our ball's motion. We just call the ball's *move( )* method and let the object itself worry about the details.

Another nice feature of OOP is that we can treat *different* objects that have *different* behaviors in a uniform manner, as long as they implement methods of the same name. For example, suppose we have a `circle` object and a `square` object. As long as both objects implement a *getArea( )* method that returns the shape's area, we can call their *getArea( )* methods without worrying about how each object calculates its own area.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

In this chapter, you'll learn how to make a basic object and how to define a category of objects (i.e., a class). Once you're comfortable with the basics, you'll learn how to use *inheritance* to share common characteristics between classes. For example, we might implement a *Horse* class that, along with our *Dog* class, is a descendant of the *Mammal* class. The *Mammal* class could implement methods and properties common to all mammals, such as the fact that they have hair, give milk, are warm-blooded, and have (up to!) four legs. Thus, inheritance can be used to create a "family tree" of related classes. Later on, you'll also learn how OOP is used to control the Flash environment through ActionScript's built-in objects and classes.

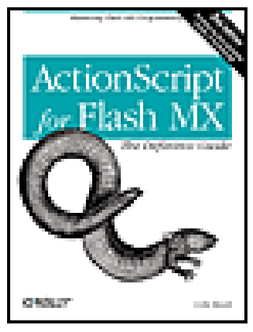
Note that not all objects represent some physical aspect of a Flash movie. At the end of this chapter, we'll show how to implement our multiple-choice quiz using OOP. We'll use objects of the *Quiz*, *Question*, and *QuizGUI* classes, of which the first two are abstract concepts, not physical things, in the movie. Our *Quiz* class, rather than bouncing movie clips around the screen, encapsulates all the behaviors of a typical quiz—adding questions, setting the title, setting the user's answer for a particular question, and so on. Even ActionScript itself uses objects to represent intangible things, such as times, dates, or the formatting applied to a text field.

Finally, bear in mind that ActionScript is quite flexible and does not force OOP on you. You can use OOP in only part of your program, or you can use it to structure everything you do. This flexibility can cause confusion for both experienced and new OOP programmers. Java developers, for example, expect everything in a program to be built as a class, which is possible, but not required in Flash. New OOP programmers, on the other hand, wonder how much OOP they should use to build what they want to build. In this chapter, we'll work towards the Java mentality, where everything in a program is represented as an object. Even if this never becomes your approach to Flash development, the exposure to rigorous OOP should enrich your understanding of objects and Flash at any level. If you're new to OOP, consider using objects whenever you create more than one of something; for example, fifteen balls, ten enemies, or two kinds of quizzes (multiple-choice or fill in the blank).

I hope that this introduction has shed some light on objects and OOP. Now let's dive in to the specifics.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir



ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Chapter 12. Objects and Classes

12.1 The Anatomy of an Object

Like an array, an object is a container of containers. A typical array holds multiple data values in individual numbered *elements*; an object, analogously, holds multiple data values in individual named *properties*. A property is, effectively, a variable defined on an object. In fact, the Java programming language calls object properties *instance variables* or simply *variables* (everything is an object in Java, so there is no need to distinguish a variable from an object property). A method, by comparison, is a function defined on an object. Practically speaking, then, an object is nothing more than a collection of related variables and functions.

Figure 12-1 depicts the properties of a sample object called `ball`. The `ball` object contains two properties: `radius` and `color`. The values of these properties are 50 and 0xFF0000 (the hex value of red). The properties are named with unique identifiers, exactly like variables. Even though each property has its own name, both are contained by the single encompassing object, `ball`.

Figure 12-1. A sample object structure

Typically, an object defines properties that are meaningfully related. More specifically, the properties of an object should be chosen such that they capture the characteristics that distinguish one instance of the object from another. *MovieClip* objects, for example, have properties specific to individual movie clips, such as their number of frames (`_totalframes`) and position on screen (`_x` and `_y`).

Traditionally, object properties are set via methods of the object to preserve the encapsulated nature of an object. That is, in strict OOP, an object should set its own properties. If an outside entity wants to set an object's property, it should do so by calling an appropriate method of the object. For example, a purist would frown on setting the `_width` property of a movie clip directly. The purist would argue that it is best to let the movie clip object set the `_width` property itself, and that code outside of the object should do so only indirectly, by calling one of the movie clip object's methods (say, *setWidth( )*). By moderating access to properties with method calls, the maintainer of the *MovieClip* class could change the name of the `_width` property to `w` without adversely affecting other users of the class.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Chapter 12. Objects and Classes

12.2 Instantiating Objects

Although it may sound backward, let's assume that we have already created an object class. This assumption isn't too ludicrous, because ActionScript provides many built-in classes, and the custom classes we'll build will behave similarly. Assuming we have an object class, we have to create a specific object *instance* (i.e., a copy) based on the class. For example, ActionScript provides the *TextField* class, but it is up to us to create individual text fields.

To create an instance of an object (i.e., to *instantiate* the object), we normally use the *new* operator with a *constructor function*, which initializes the object. The general syntax is:

```
new ConstructorFunction( )
```

Let's instantiate our first object, using a constructor function that's already built into ActionScript: the *Date( )* constructor. A *Date* object represents a particular point in time. When we instantiate a new object, we normally store the resulting instance in a variable, an array element, or a property of another object for later access. For example, the following code creates a new *Date* object and stores a reference to it in the variable named `now`:

```
var now = new Date( );
```

A specific time is a complex set of numbers that includes a second, minute, hour, day, month, and year. Using a *Date* object's methods, we can easily check or set any aspect of the time represented by the object. For example, we can invoke the *getHours( )* method to check the hour (in 24-hour format), as follows:

```
trace("The current hour is: " + now.getHours( ));
```

Let's take a closer look at how to use an object's properties and methods before showing how to add them to our own custom classes.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Chapter 12. Objects and Classes

12.3 Object Properties

As you learned earlier, properties are named data containers associated with an object. They are defined once by an object's class but set individually for each object instance. Like variables, object properties can contain any kind of data—strings, numbers, Booleans, `null`, `undefined`, functions, arrays, movie clips, or even other objects.

12.3.1 Referring to Properties

The familiar dot operator gives us access to an object's properties. We separate the name of the property from the object it belongs to using a dot (a period), as follows:

```
objectName.propertyName
```

where *`objectName`* is the name of our object and *`propertyName`* must be a legal identifier that names a property of *`objectName`*.

For example, if we have a `redBall` object instance with a `radius` property, we can retrieve the value of `radius` and assign it to a variable, `r`, as follows:

```
var r = redBall.radius;
```

Or, to set `redBall`'s `radius` property to 50, we can use:

```
redBall.radius = 50;
```

Alternatively, we can refer to a property using the `[ ]` operator, as follows:

```
objectName[propertyNameString]
```

The `[ ]` operator allows us to compose a property name using any expression that evaluates to a string. For example, here we first retrieve and then assign the value of `radius`, achieving the same results as our previous example:

```
var r = redBall["radius"];  // Retrieve
redBall["radius"] = 50;     // Assign
```

In this case, the `[ ]` operator is less concise than the dot operator and offers no apparent benefit. The `[ ]` operator is useful when the property name is a complex expression. Next, rather than using a string literal, we specify the property name with an expression that yields the string "radius":

```
var prop = "radius";        // Set a variable that contains the property name.
trace(redBall[prop]);       // The variable prop contains the
                            // property name "radius".
```

Built-in ActionScript properties are also accessed via the dot and `[ ]` operators. Here we access the value of the *Math*

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

object's `PI` property, first using the dot operator, then using the `[ ]` operator:

```
trace(Math.PI);
trace(Math["PI"]);
```

However, in pure OOP, we'll almost never access an object's properties directly; instead, we'll use methods to access property values. For example, to check the `volume` property of an instance of the built-in *Sound* class, we use:

```
trace(mySound.getVolume( ));
```

not:

```
trace(mySound.volume);  // INCORRECT!
```

Remember that the dot operator requires us to specify a property as an identifier, not a string, whereas the [ ] operator requires a string, not an identifier.

12.3.2 Using a for-in Loop to Iterate over an Object's Properties

In Chapter 8, you learned that a *for-in* loop can be used to enumerate the properties of an object. Now that you know a little more about objects, it's worth returning to the *for-in* statement briefly to review how it can be used to iterate over (i.e., access each one of) an object's properties.

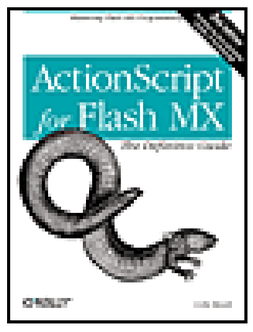
Like all loops, the *for-in* statement includes a header and a body. The body of a *for-in* statement is automatically executed once for each property in the specified object. We don't need to know the number of properties or their names in advance, because as each cycle of the loop executes, our "iterator" variable automatically becomes the name of the next property. We can therefore iterate over the properties of the object, like this:

```
// List all the properties of the ball object
for (var prop in ball) {
  trace("Property " + prop + " has the value " + ball[prop]);
}
```

Note that the iterator variable, `prop`, in this example is not an integer, as it would be in a *for* loop. That is, don't confuse a standard *for* loop, often used for accessing numbered array elements, with a *for-in* loop used to access an object's properties. For more information on *for-in* loops, including how to prevent an object's properties from being exposed to *for-in*, see Chapter 8.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir



ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Chapter 12. Objects and Classes

12.4 Object Methods

Methods are functions that are associated with objects and are typically used to perform an object-related task or access an object's data (i.e., set and retrieve its properties). We invoke methods with the function call operator, `( )`, using the dot operator to separate the method name from its object:

```
objectName.methodName( )
```

For example:

```
ball.getArea( );  // Call the getArea( ) method of ball
```

As with properties, methods are defined by a class and then used with individual object instances. For example, the *MovieClip* class defines a *play( )* method that we use on a particular movie clip instance (such as the main timeline of a movie):

```
_root.play( );
```

Methods can take parameters, return values, and otherwise behave exactly like functions, with the added feature that they can access and operate on the object through which they are invoked. For example, here we invoke *gotoAndPlay( )* as a method of the `_root` movie clip object, passing the destination frame number, 5, as an argument:

```
_root.gotoAndPlay(5);
```

And here we invoke the *getDepth( )* method, which returns an integer depth level:

```
trace(_root.getDepth( ));
```

Now that you know how to use methods and properties with objects of existing classes, we'll show how to create properties and methods for our own custom classes.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Chapter 12. Objects and Classes

12.5 Classes and Object-Oriented Programming

It's not uncommon to create dozens of complex objects that store rich information about everything from products in a shoppi cart system to bad guys with artificial intelligence in a video game. To expedite the creation of objects and to define object hierarchies (relationships between objects), we use object *classes* . A *class* is a template-style definition of an entire category objects. As you learned in the introduction to this chapter, classes describe the general features of a specific breed of objects, such as "all dogs bark."

12.5.1 Object Classes

Are you ready to get your hands dirty? Let's create a custom *Ball* class. The first step in class creation is designing the propert and methods the class supports. Our *Ball* class will have the following properties:

`radius`

: Half the diameter of a ball

`color`

: A ball's color

`xPosition`

: A ball's horizontal location

`yPosition`

: A ball's vertical location

Our *Ball* class will have no on-screen presence; for now, they will simply represent balls conceptually in our program. Later in this chapter, under Section 12.5.4, we'll consider how to actually render the ball to screen. Then, in Chapter 14, we'll show how to implement our *Ball* class as a type of movie clip. Stay tuned!

Our *Ball* class will have the following methods:

moveClipTo( )

: Used to reposition a ball

getArea( )

: Used to determine the amount of space an individual ball occupies

In natural language, our *Ball* class would be described to the interpreter as follows:

> A *Ball* is a type of object. All `ball` objects have the properties `radius` , `color` , `xPosition` , `yPosition` , which are set individually for each `ball` . All `ball` objects also share the methods *moveClipTo( )* and *getArea( )* , w are identical across all members of the *Ball* class.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

12.5.2 Making a Class

There is no specific "class declaration" device in ActionScript; there is no "class" statement that creates a new class as the *var* statement creates new variables. Instead, we define a special type of function, called a *constructor function* , that generates a r instance of our class. By defining the constructor function, we are effectively creating our class template or class definition.

Syntactically, constructor functions (or simply *constructors*) are formed just like normal functions. For example:

```
function Constructor () {
  statements
}
```

The name of a class's constructor function can be any valid function name, but, by convention, it is capitalized to indicate that a class constructor. A constructor's name should describe the class of objects it creates, as in *Ball* , *Product* , or *Vector2d* . The `statements` of a constructor function initialize the objects it creates.

An ActionScript class and an ActionScript constructor are one in the same. All instance variables are created inside the constructor; however, methods typically are not, as you'll learn shortly. Java developers should note that in ActionScript there is only one constructor per class (i.e., constructor functions cannot be *overloaded*).

We'll make our *Ball( )* constructor function as simple as possible to start. All we want it to do for now is create an empty obje To create an empty object, our *Ball( )* constructor need not include any statements.

```
// Make a Ball( ) constructor
function Ball () {

}
```

That didn't hurt much. Shortly, we'll see how to create properties inside our constructor and initialize new `ball` objects, but let's see how to generate `ball` objects using our *Ball( )* constructor function.

12.5.2.1 Creating instances of a class

A constructor function both defines our class and is used to instantiate new instances of the class. As you learned earlier in thi chapter under Section 12.2, when invoked in combination with the *new* operator, a constructor function creates, and then returns, an object instance.

Recall the general syntax for creating a new object based on a constructor function:

```
new Constructor();    // Returns an instance of the Constructor class
```

So, to create a `ball` object (an *instance*) using our *Ball* class, we write:

```
// Stores an instance of the Ball class in the variable bouncyBall
var bouncyBall = new Ball( );
```

Notice that even though our *Ball( )* constructor function is currently empty, and does not explicitly return an instance of the *B* class, when the constructor is invoked together with the *new* operator, ActionScript generates an instance automatically. Now make our *Ball* class constructor assign properties to the ball objects it creates.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

12.5.2.2 Creating a class's properties

To customize an object's properties during the object-creation stage, we use the special `this` . Within a constructor function `this` keyword is a reference to the object currently being generated. Using that reference, we can assign whatever properties like to the embryonic object. The following syntax shows the general technique:

```
function Constructor () {
  this.propertyName = value;
}
```

where `this` is the object being created, *`propertyName`* is the property we want to attach to that object, and *`value`* is data value we're assigning to that property.

Let's apply the technique to our *Ball* example. Here's how the *Ball* class constructor might assign initial values for the `radiu` `color` , `xPosition` , and `yPosition` properties to its instances. Note the use of the `this` keyword, and remember th inside our *Ball* constructor function, `this` means "the new ball being created."

```
function Ball () {
  this.radius = 10;
  this.color = 0xFF0000;
  this.xPosition = 59;
  this.yPosition = 15;
}
```

> When creating object properties, the *var* keyword (used to create variables) must not be used. The following code generates a compile-time error:
>
> ```
> var this.radius = 10;
> ```

With the *Ball( )* constructor thus prepared, we can create *Ball* instances (i.e., objects of the *Ball* class) bearing our predefined properties—`radius` , `color` , `xPosition` , and `yPostion` —by invoking *Ball( )* with the *new* operator, just as we di earlier. For example:

```
// Make a new instance of Ball
var bouncyBall = new Ball( );
```

Now we can access the properties of `bouncyBall` that were set when it was made by the *Ball( )* constructor, as follows:

```
trace(bouncyBall.radius);     // Displays: 10
trace(bouncyBall.color);      // Displays: 16711680 (decimal for 0xFF0000)
trace(bouncyBall.xPosition);  // Displays: 59
trace(bouncyBall.yPosition);  // Displays: 15
```

Now wasn't that fun?

Unfortunately, so far our *Ball( )* constructor uses fixed values when it assigns properties to the objects it creates (e.g., the initi radius is always 10). Every object of the *Ball* class, therefore, will have the same initial property values, which is very inflexi Let's add parameters to our class constructor that let us specify varying initial property values, so that each ball can be unique when it is born.

Here's the general syntax used to define parameters for a class constructor:

```
function Constructor (value1, value2, value3) {
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

```
  this.property1 = value1;
  this.property2 = value2;
  this.property3 = value3;
}
```

Inside the *Constructor* function, we refer to the object being created with the `this` keyword, just as we did earlier. This time however, we don't hardcode the object properties' values. Instead, we assign the values of the arguments *value1* , *value2* , and *value3* to the object's properties. When we want to create a new, unique instance of our class, we pass our class constructor function the initial property values for our new instance:

```
someObject = new Constructor (value1, value2, value3);
```

Let's see how this applies to our *Ball* class, shown in Example 12-1.

Example 12-1. A ball class with properties

```
// Make the Ball( ) constructor accept property values as arguments.
function Ball (radius, color, xPosition, yPosition) {
  this.radius = radius;
  this.color = color;
  this.xPosition = xPosition;
  this.yPosition = yPosition;
}

// Invoke our constructor twice, passing it arguments to use as
// our new objects' property values.
var greenBall = new Ball(42, 0x00FF00, 59, 15);
var whiteBall = new Ball(14, 0xFFFFFF, 50, 12);

// Now let's see if it worked...
trace(greenBall.radius);  // Displays: 42,  :) Pretty cool...
trace(whiteBall.radius);  // Displays: 14
                          // Each instance has unique property values.
```

By convention, the parameters being passed in (`radius` , `color` , etc.) use names similar to the corresponding internal properties (`this.radius` , `this.color` , etc.), but don't confuse the two. The former are transient parameter variables used simply to transmit the arguments passed to the constructor function; the latter (beginning with `this`) are the object properties where those values are permanently stored.

In general, a class's properties should all be declared at the top of the class constructor. Typically, a new property should not b added to a class by a method. Unlike Java, ActionScript leniently allows methods to create new object properties, but doing s leads to unmanageable class structures. Methods should only *modify* properties declared in the constructor. However, for the sake of reduced memory consumption, a class with, say, 100 properties can be forced to relegate property creation to methods instantiating instances has a noticeable effect on Flash Player performance.

We're almost done building our *Ball* class. But you'll notice that we're still missing the *moveClipTo( )* and *getArea( )* methods discussed earlier. Let's take care of that now.

12.5.2.3 Creating a class's methods

Methods are added to a class by assigning a function to the special `prototype` property of the class constructor (we'll cove the technical role of `prototype` later in this chapter under Section 12.7). The general syntax is:

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

```
Constructor.prototype.methodName = function;
```

where *Constructor* is the class constructor function, *methodName* is the name of the method, and *function* is t function executed when the method is used. The *function* can be supplied as a function reference or a function literal. Fc example:

```
// Function reference
Ball.prototype.getArea = calcArea;
function calcArea ( ) {
  // ...method body goes here...
}

// Function literal
Ball.prototype.getArea = function () {
  // ...method body goes here...
};
```

The function literal approach is generally preferred for methods because it does not leave an unneeded function reference in memory after method assignment is finished.

Java developers should note that the datatypes of a method's return value and parameters need not be specified in ActionScript. Similarly, a method's access level cannot be set, nor is there any syntactic facility for creating static methods (though we'll learn how to simulate them later).

You learned earlier that a method of an object often retrieves or sets the properties of the object on which it is invoked. So, th question is, how does the method refer to that object in order to access its properties? Answer: through the magic of the `this` keyword. Remember that we used the `this` keyword inside our *Ball* constructor function to refer to new ball objects being created by the constructor. Inside a method, `this` similarly refers to the ball object currently being operated on. For example access an individual *Ball* instance's `radius` property inside *getArea( )* , we use `this.radius` , as shown in the followin listing for our *Ball.getArea( )* method:

```
Ball.prototype.getArea = function ( ) {
  // The this keyword means the ball on which getArea( ) was invoked.
  return Math.PI * this.radius * this.radius;
};
```

Likewise, a method can set the value of a property via `this` , as shown in our *Ball* class's *moveClipTo( )* method. The *moveClipTo( )* method assigns new values to the `xPosition` and `yPosition` properties of a ball. Notice that the *moveClipTo( )* method takes two parameters, `x` and `y` , which specify the ball's new position:

```
Ball.prototype.moveClipTo = function (x, y) {
  // Notice the use of this to refer to the ball being moved.
  this.xPosition = x;
  this.yPosition = y;
};
```

We've now created the all properties and methods of our *Ball* class, so our work is done! The final code is shown in Example 2. Attach the example code to frame 1 of a new movie; then choose Control Test Movie to see it in action.

Example 12-2. The completed Ball class

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

```
// The Ball( ) class constructor.
function Ball (radius, color, xPosition, yPosition) {
  // Create Ball properties.
  this.radius = radius;
  this.color = color;
  this.xPosition = xPosition;
  this.yPosition = yPosition;
}
// The Ball.getArea( ) method.
Ball.prototype.getArea = function () {
  // The keyword this means the ball on which getArea( ) was invoked.
  return Math.PI * this.radius * this.radius;
};

// The Ball.moveClipTo( ) method.
Ball.prototype.moveClipTo = function (x, y) {
  // Notice the use of this to refer to the ball being moved.
  this.xPosition = x;
  this.yPosition = y;
};
// Make a large red ball, positioned at the origin (0, 0).
var bigBall = new Ball(600, 0xFF0000, 0, 0);

// Check the ball's area, which is based on its radius.
trace(bigBall.getArea( ));  // Displays: 1130973.35529233

// Reposition our ball via the moveClipTo( ) method.
bigBall.moveClipTo(100, 200);

// Check the ball's new xPosition.
trace(bigBall.xPosition);  // Displays: 100
```

Invoking a method and defining a method have similar syntax but very different results. It's crucial to understand the differen
When *defining* a method, *omit* the parentheses:

```
SomeClass.prototype.someMethod = someFunction;
```

When *invoking* a method, *include* the parentheses:

```
someObj.someMethod( );
```

The difference in syntax is subtle. Make sure to get it right.

12.5.2.4 Getter and Setter Methods

As you learned earlier, strict OOP practice frowns upon accessing an object property directly. Instead, so-called getter/setter methods should be used to access the property. A *getter method* retrieves the value of the property, while a *setter method* assi it. For example, our *Ball* class defines a `radius` property. Rather than access that property directly, it's considered good for access it with a *getRadius( )* method and a *setRadius( )* method. For example, consider this simplified version of our *Ball* clas

```
// The Ball class
function Ball (radius) {
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

```
  // Instance properties
  this.radius = 0;

  // Initialization
  this.setRadius(radius);
}

Ball.prototype.getRadius = function ( ) {
  return this.radius;
}

Ball.prototype.setRadius = function (newRadius) {
  this.radius = newRadius;
}

Ball.prototype.getArea = function () {
  return Math.PI * this.getRadius( ) * this.getRadius( );
};

// A ball instance
var largeBall = new Ball(400);
largeBall.setRadius(500);
trace(largeBall.getArea( ));  // Displays: 78.5398163397448
```

The class constructor declares the `radius` property and suggests its datatype by assigning it a number (0). However, all oth access to `radius` is mediated by the *getRadius( )* and *setRadius( )* methods. Even the constructor itself uses *setRadius( )* to assign `radius` its starting value. Inside the *getArea( )* method, the `radius` property is retrieved via calls to *getRadius( )* . in the outside world, we use *setRadius( )* to assign the `largeBall` 's new radius. By using getter/setter methods, we indicat that we want to change some attribute of the ball, but we let the object itself worry about how that change is implemented. Changing the internal implementation from, say, using a `radius` property measured in pixels to using a `radius` property measured in centimeters won't break any code that properly uses the getter and setter methods; only the getter/setter methods to be modified. If code outside the object refers to the `radius` property directly, the programmer maintaining the object has freedom to change it.

Getter/setter methods safely encapsulate the workings of an object, but they are admittedly more verbose than a property assignment (and often make the code harder to read). Therefore, in practice, many programmers don't bother implementing getter/setter methods, and many programmers don't use them even when they are available. To address this issue, Flash MX added the *addProperty( )* method to all objects (derived from the *Object* class). The *addProperty( )* method creates a so-called *getter/setter property* that, when accessed, automatically invokes the appropriate getter/setter method. This lets programmers and retrieve object properties with somewhat more brevity, while still maintaining the benefits of pure OOP practice. The following example shows the general technique. For much more information, see *Object.addProperty( )* in the ActionScript Language Reference.

```
// The Ball class
function Ball (radius) {
  // Instance properties.
  this._radius = 0;

  // Initialization
  this.setRadius(radius);
}
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

```
Ball.prototype.getRadius = function ( ) {
  return this._radius;
}

Ball.prototype.setRadius = function (newRadius) {
  this._radius = newRadius;
}

Ball.prototype.getArea = function () {
  return Math.PI * this.radius * this.radius;
};

// The getter/setter radius property, defined on prototype
Ball.prototype.addProperty("radius",
                           Ball.prototype.getRadius,
                           Ball.prototype.setRadius);

// A ball instance
var largeBall = new Ball(400);
largeBall.radius = 5;         // Invokes the setRadius( ) method automatically.
trace(largeBall.getArea( ));  // Displays: 78.5398163397448
```

Note that the class uses an internal `_radius` property to store the value of the getter/setter property called `radius` . The internal property must not have the same name as the externally visible getter/setter property, and is therefore prefaced with a underscore by convention. Getter/setter properties can be used to create read-only properties, as discussed under *Object.addProperty( )* in the ActionScript Language Reference.

Reader Exercise: Try adding *getColor( )* and *setColor( )* methods that get and set the `color` property in the *Ball* class. Once get that far, try using *Object.addProperty( )* to turn `color` into a getter/setter property. Don't forget to use an internal `_colo` property to store the value of the getter/setter property called `color` !

12.5.2.5 Alternate means of method assignment

It's possible, though not advised, to add a method to each instance by assigning a function to an instance property within the c constructor. Here's the generic syntax:

```
function Constructor () {
  this.methodName = function;
}
```

Assigning a method on a per-instance basis in a class constructor is possible in ActionScript because an ActionScript method technically just a function stored in an object property. The `function` assigned as the new method can be supplied in seve ways, as shown in the following examples. The `function` can be a function literal, as in:

```
function Ball (radius) {
  this.radius = radius;
  // Add the getArea( ) method...
  this.getArea = function () { return Math.PI * this.radius * this.radius; };
}
```

Alternatively, the `function` can be declared *inside* the constructor:

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

```
function Ball (radius) {
  this.radius = radius;
  // Add the getArea( ) method...
  this.getArea = calcArea;
  function calcArea () {
    return Math.PI * this.radius * this.radius;
  }
}
```

Finally, the `function` can be declared *outside* the constructor but assigned *inside* the constructor:

```
// Declare the getArea( ) function
function calcArea () {
  return Math.PI * this.radius * this.radius;
}

function Ball (radius) {
  this.radius = radius;
  // Add the getArea( ) method...
  this.getArea = calcArea;
}
```

There's no real difference between the three approaches. Function literals often prove the most convenient to use, but they are perhaps not as reusable as defining a function outside of the constructor so that it can also be used in other constructors.

Regardless, all three approaches lack efficiency and are not recommended. When we assign a method to the objects of a class using the techniques we've just seen, we are unnecessarily duplicating the method on *every instance* of our class. The *getArea* formula is the same for every `ball` , and hence the code that performs this task should be stored in a central location—the constructor's `prototype` property. For better or worse, ActionScript is not a strict language; it often provides several ways accomplishing a similar goal.

12.5.2.6 Static Methods and Properties

In Java, a *static method* (also called a *class method*) is a method that is invoked through a class directly, rather than through a object (i.e., an instance of a class). Similarly, a *static variable* in Java is akin to a property that is accessed through a class, independent of any object instance. Java's static methods and properties are used to store behavior and information that logically relates to a class but should be universally available, rather than available only through a specific object instance. For example, *Square* class might define a static property, `numSides` , that indicates the number of sides for all square instances. By convention, the getter/setter methods for a static property—such as *getNumSides( )* and *setNumSides( )* —would be implemented as static methods. ActionScript's built-in *TextField* class defines the static method *getFontList( )* , which returns a list of fonts on the user's system. The *getFontList( )* method is logically associated with text fields in general, not a specific text field, so it is implemented as a static method.

ActionScript does not have a syntactic mechanism for explicitly declaring methods or properties as static (as Java does). Instead, static methods and properties are created by attaching properties and methods to a class's constructor function, as follows:

```
// Class definition
function SomeClass ( ) {
  statements
}

// Static Property
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

```
SomeClass.someStaticProperty = value;

// Static Method
SomeClass.someStaticMethod = function;
```

Note that, unlike Java, ActionScript's static methods have no limitation on the variables, properties, or methods they can acce (in Java, a static method can refer only to static variables and methods). Furthermore, in a static ActionScript method, the `th` keyword refers to the class constructor function, whereas Java's static methods cannot use the `this` keyword.

When defining a property that will hold a constant value, it is also perfectly legitimate to use the class's `prototype` proper as follows:

```
function Square ( ) {
  statements
}
Square.prototype.numSides = 4;
```

However, this approach demands that `numSides` be accessed through an instance of *Square* , not through the *Square* class directly. If you want to use a property without first creating an instance of a class, you should make that property static.

Sometimes, static properties and methods are used to package a series of related constants and functions into a class that will never be instantiated. In ActionScript, this is normally achieved by adding properties and methods to a generic instance of the *Object* class. For example, the *Math* , *Key* , *Mouse* , and *Stage* objects are implemented in this manner.

For the benefit of Java programmers, Table 12-1 outlines the rough equivalencies between Java and ActionScript methods an properties.

Table 12-1. Java and ActionScript property equivalencies

Java name	Java description	ActionScript description	ActionScript example

Instance variables	Variables local to an object instance	Properties defined in a class constructor function and copied to objects	`function Square (side) {`
			`  this.side = side;`
			`}`

Instance methods	Methods invoked on an object instance	Methods defined on the `prototype` object of a class constructor function and accessed automatically through `prototype` when invoked on an instance	`Square.prototype.getArea = function ( ) { ...`
			
			`mySquare.getArea();`

Class variables	Variables with the same value across all object instances of a class	Properties defined on a class constructor function	`Square.numSides = 4;`
Class methods	Methods invoked through a class	Methods defined on a class constructor function	`Square.findSmaller =` `function (square1, square2) { ... }`

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

12.5.3 Inheritance: Superclasses and Subclasses

One of the crucial features of classes in advanced object-oriented programming is their ability to inherit properties and metho That is, an entire class can adopt the properties and methods of another class and add its own specialized features. The class tl borrows the properties and methods of another class is known as the *subclass* . The class that has the properties and methods borrowed is known as the *superclass* . The subclass is said to *extend* the superclass because it borrows (*inherits*) the supercla methods and properties but also adds its own. Several subclasses can inherit from the same superclass; for example, there may a *Car* superclass with *Coupe* and *Sedan* subclasses. The ability of classes to adopt methods and properties from other classes i known as *inheritance* .

For example, suppose we want to create two types of quiz programs: a multiple-choice quiz and a fill-in-the-blank quiz. The t programs will have much in common. They'll both load questions, display questions, accept answers, and tally the user's scor However, there will be minor variations in the ways they present questions, receive answers, and tally the score. Rather than creating two highly similar programs separately, we create a general *Quiz* class that defines methods and properties common both quizzes. Then, to represent our specific quiz types, we create two *Quiz* subclasses, *MultipleChoiceQuiz* and *FillInBlank* . The subclasses inherit all of *Quiz* 's abilities but add methods and properties specific to the tasks of multiple-choice or fill-in the-blank question management. The subclasses can even replace (or *override*) the behavior of one or more *Quiz* methods.

Let's see how to implement inheritance in ActionScript with our simple *Ball* example class. Earlier we created a *getArea( )* method on *Ball* . Now suppose that we want to create many different types of circular objects. Rather than creating a *getArea* method on each circular object class, we create a general superclass, *Circle* , that defines a general method, *getArea( )* , used t calculate the area of any circular object. Then, we make *Ball* (and any other circular object class) a subclass of *Circle* , so *Ball* inherits the *getArea( )* method from *Circle* . Notice the hierarchy—the simplest class, the *Circle* superclass, defines the most general methods and properties. Another class, the *Ball* subclass, builds on that simple *Circle* class, adding features specific t *Ball* -class instances but relying on *Circle* for the basic attributes that all circular objects share.

12.5.3.1 Making a superclass

Earlier you learned that methods are defined on the special `prototype` property of a class's constructor function.

> To specify a *superclass* for a given class, we completely replace the class's `prototype` property with a new instance of the desired superclass.

We'll delay our technical consideration of the `prototype` property until later. For now, concentrate on the general syntax t to specify a superclass:

```
SubclassConstructor.prototype = new SuperClassConstructor( );
```

where `SubclassConstructor` is the subclass's constructor function and `SuperClassConstructor` is the superclass's constructor function. Example 12-3 applies the technique to our *Ball* and *Circle* example.

Example 12-3. Creating a superclass

```
// Create a Circle (superclass) constructor
function Circle () {
  // Instance properties and initialization go here.
}
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

```
// Create the Circle.getArea( ) method
Circle.prototype.getArea = function ( ) {
  return Math.PI * this.radius * this.radius;
};

// Create our usual Ball class constructor
function Ball (radius, color, xPosition, yPosition) {
  this.radius = radius;
  this.color = color;
  this.xPosition = xPosition;
  this.yPosition = yPosition;
}

// Make Ball a subclass of Circle
Ball.prototype = new Circle( );

// Now let's make an instance of Ball and check its properties
var bouncyBall = new Ball (16, 0x445599, 34, 5);
trace(bouncyBall.xPosition);   // 34
trace(bouncyBall.getArea( ));   // 804.247719318987
                               // The getArea( ) method was inherited from Circ
```

By default, new objects inherit directly from the generic *Object* class. We set a class constructor function's `prototype` property to make the class inherit from a class other than the generic *Object* class. We'll learn precisely how this works later in this chapter under Section 12.7.

The class hierarchy in Example 12-3 has poor structure—*Ball* defines `radius`, but `radius` is a property common to all circles, so it belongs in our *Circle* class. The same is true of `xPosition` and `yPosition`. To fix the structure, we'll mov `radius`, `xPosition`, and `yPosition` to *Circle*, leaving only `color` in *Ball*. (For the sake of the example, we'll tre `color` as a property only balls can have.)

Conceptually, here's the setup of our revised *Circle( )* and *Ball( )* constructors (the *Circle.getArea( )* method is not shown):

```
// Create the Circle (superclass) constructor
function Circle (radius, xPosition, yPosition) {
  this.radius = radius;
  this.xPosition = xPosition;
  this.yPosition = yPosition;
}

// Create the Ball class constructor
function Ball (color) {
  this.color = color;
}
```

Having moved the properties around, we're faced with a new problem. How can we provide values for `radius`, `xPositi`, and `yPosition` when we're creating objects using *Ball* and not *Circle*? We have to make one more adjustment to our *Ba* constructor code. First, we set up *Ball* to receive all the required properties as parameters:

```
function Ball (color, radius, xPosition, yPosition) {
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Next, within our *Ball( )* constructor, we use the *super* operator to invoke the *Circle* 's constructor function on the *Ball* object b constructed. While we're at it, we pass *Circle* 's constructor the arguments that were supplied to *Ball* :

```
super(radius, xPosition, yPosition);
```

Our completed class/superclass code is shown in Example 12-4.

Example 12-4. A class and its superclass

```
// Create a Circle (superclass) constructor
function Circle (radius, xPosition, yPosition) {
  this.radius = radius;
  this.xPosition = xPosition;
  this.yPosition = yPosition;
}

// Create the Circle.getArea( ) method
Circle.prototype.getArea = function ( ) {
  return Math.PI * this.radius * this.radius;
};

// Create our Ball class constructor
function Ball (color, radius, xPosition, yPosition) {
  // Invoke the Circle( ) constructor on the ball object, passing values
  // supplied as arguments to the Ball( ) constructor
  super(radius, xPosition, yPosition);
  // Set the color of the ball object
  this.color = color;
}

// Set Ball's superclass to Circle
Ball.prototype = new Circle( );

// Now let's make an instance of Ball and check its properties
var bouncyBall = new Ball (0x445599, 16, 34, 5);
trace(bouncyBall.xPosition);   // 34
trace(bouncyBall.getArea());   // 804.247719318987
trace(bouncyBall.color);       // 447836
```

In Flash 5, the *super* operator is not available, so we must replace this line:

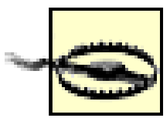
```
  super(radius, xPosition, yPosition);
```

with these three lines, which accomplish the same thing.

```
  this.superClass = Circle;
  this.superClass(radius, xPosition, yPosition);
  delete this.superClass;
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir



When a class's superclass is set using the standard technique shown in Example 12-4, all existing methods defined on the `prototype` property of the subclass constructor are destroyed! Make sure to always set a subclass's superclass *before* creating the subclass's methods!

12.5.3.2 Accessing overridden methods

When a subclass implements a method by the same name as a method of its superclass, the subclass version overrides the superclass version. That is, instances of the subclass use the subclass's implementation of the method, not the superclass's, if i exists. However, within a subclass method, the *super* operator gives us access to the superclass version of any method. For example, suppose both *Circle* and *Ball* define a *toString( )* method, as follows:

```
Circle.prototype.toString = function ( ) {
  return "A circular object";
}

Ball.prototype.toString = function ( ) {
  return "A ball object";
}
```

By default, *Ball* instances will use the *Ball* class's *toString( )* method and ignore the method of the same name in *Circle* . However, we can create another *Ball* method, named *superToString( )* , that specifically accesses *Circle.toString( )* , as follow

```
Ball.prototype.superToString = function ( ) {
  return super.toString( );
}
```

See more about the *super* operator in Chapter 5.

12.5.3.3 Polymorphism

Inheritance makes another key OOP concept, *polymorphism* , possible. Polymorphism is a fancy word meaning "many forms simply means that you tell an object what to do but leave the details up to the object. It is best illustrated with an example. Suppose you are creating a cops and robbers game. There are multiple cops, robbers, and innocent bystanders displayed on th screen simultaneously, all moving independently according to different rules. The cops chase the robbers, the robbers run awa from the cops, and the innocent bystanders move randomly, confused and frightened. In the code for this game, suppose we create an object class to represent each category of person:

```
function Cop () { ... }
function Robber () { ... }
function Bystander () { ... }
```

In addition, we create a superclass, *Person* , from which the classes *Cop* , *Robber* , and *Bystander* all inherit:

```
function Person () { ... }
Cop.prototype       = new Person();
Robber.prototype    = new Person();
Bystander.prototype = new Person();
```

On each frame of the Flash movie, every person on the screen should move according to the rules for his class. To make this happen, we define a *move( )* method on every object (the *move( )* method is customized for each class):

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

```
Person.prototype.move = function () { ... default move behavior ... }
Cop.prototype.move = function () { ... move to chase robber ... }
Robber.prototype.move = function () { ... move to run away from cop ... }
Bystander.prototype.move = function ()  { ... confused, move randomly ... }
```

On each frame of the Flash movie, we want every person on the screen to move. To manage all the people, we create a master array of *Person* objects. Here's an example of how to populate the `persons` array:

```
// Create our cops
var cop1 = new Cop();
var cop2 = new Cop();

// Create our robbers
var robber1 = new Robber();
var robber2 = new Robber();
var robber3 = new Robber();

// Create our bystanders
var bystander1 = new Bystander();
var bystander2 = new Bystander();

// Create an array populated with cops, robbers, and bystanders
persons = [cop1, cop2, robber1, robber2, robber3, bystander1, bystander2];
```

In every frame of the Flash movie, we call the *moveAllPersons( )* function, which is defined as follows:

```
function moveAllPersons () {
  for (var i=0; i < persons.length; i++) {
    // Call the move() method for each person in the array
    persons[i].move();
  }
}
```

When *moveAllPersons( )* is invoked, each cop, robber, and bystander will move according to the individual rules associated with his class, as defined by its *move( )* method. This is polymorphism in action—objects with common characteristics can be organized together, but they retain their individual identities. The cops, robbers, and bystanders have much in common, embodied by the superclass *Person* . They have operations in common, such as movement. However, they can implement the common operations differently, and they can support other class-specific operations and data.

> Polymorphism permits dissimilar objects to be treated uniformly. We can use the *move( )* function to cause each object to move, even though the *move( )* function for each class is unique. In fact, even if the *Cop* , *Robber* , and *Bystander* classes do not inherit from *Person* , we can still use their *move( )* method in a uniform manner.

12.5.3.4 Determining whether an object belongs to a superclass

Suppose we have a *Rectangle* class that is a subclass of *Shape* . To check whether an object is a descendant of *Shape* , we can use the *instanceof* operator, introduced in Flash MX:

```
rect = new Rectangle(15, 35);
trace(rect instanceof Rectangle);  // Yields true
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

```
trace(rect instanceof Circle);     // Yields false
trace(rect instanceof Shape);      // Yields true
```

We can use *instanceof* to operate conditionally on an object, depending on its class. The following sample, taken from the qui the end of this chapter, deletes all movie clips and text fields in `quizClip` . It cycles through `quizClip` 's properties and invokes the appropriate removal method on each, depending on whether it is an instance of the *TextField* class or belongs to t *MovieClip* datatype.

```
for (var p in quizClip) {
  if (typeof quizClip[p] =  = "movieclip") {
    quizClip[p].removeMovieClip( );
  } else if (quizClip[p] instanceof TextField) {
    quizClip[p].removeTextField( );
  }
}
```

In Flash 5, the *instanceof* operator doesn't exist, so we must use the chain of `prototype` objects (called the *prototype chai* to determine an object's class, as shown in Example 12-5. We'll learn more about `prototype` objects and `_ _proto_ _` shortly.

Example 12-5. Walking the prototype chain

```
// This function checks if theObj is a descendant of theClass
function objectInClass (theObj, theClass) {
  while (theObj._ _proto_ _ != null) {
    if (theObj._ _proto_ _ =  = theClass.prototype) {
      return true;
    }
    theObj = theObj._ _proto_ _;
  }
  return false;
}

// Make a new instance of Rectangle
myObj = new Rectangle();

// Now check if myRect inherits from Shape
trace (objectInClass(myRect, Shape)); // Displays: true
```

12.5.3.5 The end of the inheritance chain

All objects descend from the top-level *Object* class. Hence, all objects inherit the properties defined in the *Object* constructor example, the *toString( )* and *valueOf( )* methods). We can, therefore, add new properties to *every* object in a movie by adding new properties to the *Object* class. Properties attached to `Object.prototype` will proliferate throughout the entire class hierarchy, including all built-in objects, such as *Array* , *TextField* , and even *MovieClip* objects. This approach can be used to make a sort of global variable or function. In the following code, for example, we hardcode the Stage width and height dimensions into the *Object* class `prototype` . We can then access that information from any movie clip (this technique wa common in Flash 5 movies, although in Flash Player 6, the Stage dimensions are available as `Stage.width` and `Stage.height`):

```
Object.prototype.mainstageWidth  = 550;
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

```
Object.prototype.mainstageHeight = 400;
trace(anyClip.mainstageWidth);              // Displays: 550
```

`Object.prototype` properties are inherited not only by movie clips, but also by every object, so even our own custom objects and instances of built-in classes such as *XML* , *Date* , and *Sound* inherit properties assigned to `Object.prototy` This side-effect causes unwanted properties to appear in *for-in* loops. Therefore, when exporting for Flash Player 6, *always* u the `_global` object to create global properties and methods. See `_global` in the *Language Reference* for details.

Reader Exercise: We can attach new properties and methods to any built-in class. Try adding the case-insensitive alphabetical sort function from Example 11-6 to the *Array* class as a custom method.

12.5.4 Displaying the Ball on Screen

Throughout this chapter, we've followed the bouncing *Ball* class through a lot of theory. However, for all our work, we still haven't actually made a single ball appear on screen. That's because updating the display isn't necessarily the *Ball* class's conc Drawing the current content of the screen is often the responsibility of a separate class that deals specifically with the screen a graphical user interface. How the screen is drawn, when it is drawn, and what draws the screen can vary greatly from applicat to application. For example, in our Quiz at the end of this chapter, we'll update the screen from a *QuizGUI* class every time a question is displayed. But, in other applications, a ball object would probably move around a lot, so it would need to be displa repeatedly via either *MovieClip.onEnterFrame( )* or *setInterval( )* .

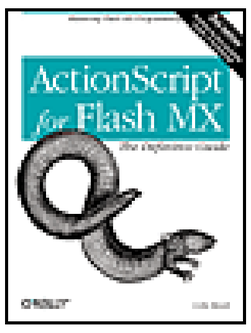
Here are the major issues to consider when designing an application's display system.

- First decide whether the display elements will render themselves or be rendered by a central class. For example, will th *Ball* class provide its own *drawBall( )* method, or will there be a *drawBalls( )* or *drawScreen( )* method on some other class?
- Decide how often the elements should be displayed—either when some event occurs (such as a mouse click) or repeate (as fast as frames are displayed in the Flash Player).
- Decide on the rendering technique. Each visual element in a Flash movie must be displayed in a movie clip. However, movie clips can be placed manually at authoring time, attached with ActionScript at runtime, or even created from scra using the MovieClip Drawing API.
- Decide on a screen refresh strategy. Will each visual element be maintained as a single persistent movie clip that is upd regularly, or will the entire contents of the Stage be cleared and recreated each time the display is updated?

These decisions will dictate exactly how an application manages its visual assets. For example, we might have a *BallSimulati* class that updates the screen by drawing circles that represent a list of ball objects. Or, we might add a *Ball* property, `mc` , tha holds a movie clip representing each *Ball* object. A *Ball.drawBall( )* method could then update the display of a *Ball* object's corresponding movie clip. When an object and the movie clip that represents it are tied that tightly together, it's often wise to implement the class as a *MovieClip* subclass, as described in Chapter 14.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir



ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Chapter 12. Objects and Classes

12.6 Using Standalone Object Instances as Associative Arrays

As we saw earlier, ActionScript leniently allows methods to assign new properties to objects. In fact, ActionScript is even mo flexible than that—it allows any code to create a new property or method on a single object instance (rather than on all instan of a class). For example, the following code creates an instance, `obj` , of the generic *Object* class and immediately assigns it new properties `x` (set to 10) and `y` (set to 20):

```
var obj = new Object( );
obj.x = 10;
obj.y = 20;
```

Assigning properties and methods to individual objects is, of course, bad OOP practice. But in ActionScript, it is also a completely legitimate way to store a group of named values (much like a hash or associative array). In fact, many of ActionScript's built-in methods take generic objects as parameters. For example, *MovieClip.globalToLocal( )* takes a *point* ob parameter that must supply an `x` property and a `y` property. The `x` and `y` properties represent a global position that will be converted to a movie clip's local coordinate system.

```
pt = new Object( );    // Create generic object to hold our point
pt.x = 0;              // Left border of main Stage
pt.y = 0;                          // Top border of main Stage
this.globalToLocal(pt);           // Convert pt to local coordinates
trace("From the current clip, the top-left corner of the main Stage is at ");
trace("x: " + pt.x + "y: " + pt.y);
```

Similarly, *MovieClip.attachMovie( )* takes an *`initObj`* parameter used to transfer an object's properties to a new movie cl instance, `car_mc` , as follows:

```
// Create an object
var initObj = new Object( );

// Create properties on the object
initObj.speed = 3;
initObj.onEnterFrame = function ( ) {
  this._x += this.speed;
};

// Transfer the object's properties to car_mc
this.attachMovie("CarSymbol", "car_mc", 1, initObj);
```

Notice that this code defines *onEnterFrame( )* as a method on *`initObj`* , which, given the task at hand, is perfectly legal (a convenient!).

12.6.1 Object Literals

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

To create a generic object of the *Object* class directly in our code without using the *Object* constructor, we can use an object literal, just as we might use a string literal or an array literal. An object literal is a series of comma-separated property name/v pairs, enclosed in curly braces. Here's the general syntax:

```
{ property1: value1, property2: value2, property3: value3 }
```

The names of properties in an object literal must be legal identifiers, as described in Chapter 14. The values can be any valid expression. For example:

```
// An object literal with two numeric properties
myObject = { x: 30+10, y: 20*2 };
```

Object literals are used for the sake of terseness. For example, in the following code we transfer the `speed` and `onEnterFrame` properties to `car_mc` more succinctly (though perhaps less legibly) than we did earlier:

```
this.attachMovie("CarSymbol", "car_mc", 1, {speed:3, onEnterFrame: function (
                                                    this._x += this.speed;}})
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Chapter 12. Objects and Classes

12.7 The Almighty Prototype Chain

In Chapter 2, we saw that the interpreter uses a scope chain to resolve (look up the value of) unqualified variable references. T resolve property values, the interpreter uses a *prototype chain* . We've already encountered the `prototype` property in our examination of methods and superclasses. Now let's take a look "under the hood" of ActionScript's inheritance system to see l `prototype` property really works. For the sake of the current discussion, the term *property* refers to object properties, meth event handlers (the latter two of which are, technically, functions stored in properties).

As we've seen, every class constructor function is automatically given a property called `prototype` . For example, when w a *Book* class, the *Book* constructor automatically defines `Book.prototype` :

```
function Book ( ) {
}
trace(Book.prototype);  // Displays: [object Object]
```

Initially, a constructor function's `prototype` property contains a generic object (i.e., an instance of the generic *Object* class Earlier, we added methods to the `prototype` object that were accessible to all instances created by the constructor. How d work?

Suppose we define a property directly on an object, either outside of a constructor, like this:

```
var importantDates = new Object( );
// Define gwenBirthday property.
var importantDates.gwenBirthday = "January 17";
```

or inside a constructor, like this:

```
function Book (author) {
  // Define author property.
  this.author = author;
}
var asGuide = new Book("Colin Moock");
```

When we ask the interpreter to retrieve any property, it first checks if the object, itself, defines the property. For example, in t following code, the interpreter checks `asGuide` for the `author` property, finds it, and displays its value in the Output win

```
trace(asGuide.author);  // Displays: Colin Moock
```

But if the interpreter cannot find the specified property on the object instance, it searches the object's class constructor's `prototype` property. For example, suppose we define a property, `description` , on `Book.prototype` :

```
Book.prototype.description = "Bound paper with writing on it.";
```

Then we check the value of `asGuide.description` :

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

```
trace(asGuide.description);  // Displays: Bound paper with writing on it.
```

The interpreter can't find `description` directly on `asGuide` , so it looks for it on `Book.prototype` . `Book.prot` defines `description` , so the interpreter uses it as the value of `asGuide.description` . But if we add a `descri` property directly to an instance of *Book* , either inside or outside the *Book* constructor, the instance property overrides the `Book.prototype` property. For example:

```
function Book (author, desc) {
  // Define author property
  this.author = author;
  // Define description property
  this.description = desc;
}
Book.prototype.description = "Bound paper with writing on it.";

var asGuide = new Book("Colin Moock", "A book about ActionScript");
trace(asGuide.description);  // Displays: "A book about ActionScript"
                             // NOT: "Bound paper with writing on it."
```

Let's see how this applies to methods by adding a *getAuthor( )* method to *Book* :

```
Book.prototype.getAuthor = function ( ) {
  return this.author;
}
```

When we call *asGuide.getAuthor( )* , the interpreter checks `asGuide` to see if it defines a property named `getAuthor` . I not, so the interpreter checks `Book.prototype` for `getAuthor` . There the interpreter finds our custom method and ex

What would have happened if `Book.prototype` had not defined `getAuthor` ? Astute readers will recall that, by defau `Book.prototype` stores a generic object instance. When the interpreter cannot find a property on that generic instance, it the instance's *own* class constructor's `prototype` property—which is, you guessed it, `Object.prototype` . That's ho prototype chain works: when a property is not found on an object instance, the object's class's `prototype` is searched. If th property is not there, the `prototype` object's own class's `prototype` is searched, and so on up the chain, until `Object.prototype` is reached. If `Object.prototype` does not define the property, the property is `undefined` the `prototype` chain for our `asGuide` instance (note that prototype chains, like scope chains, are read from the bottom u

```
Object.prototype      // If not found here, return undefined
Book.prototype        // If not found here, look on Object.prototype
asGuide               // If not found here, look on Book.prototype
                      // INTERPRETER STARTS LOOKING FOR PROPERTIES HERE
```

When a property name appears more than once in the prototype chain, the lowest property value on the chain is used (as we s earlier with the `description` property). The lower property is said to *override* the higher property. For example, the *Obje* defines a *toString( )* method on `Object.prototype` . We can override that version of *toString( )* for our *Book* class by *toString( )* on `Book.prototype` :

```
Book.prototype.toString = function ( ) {
  return this.description + ", by " + this.getAuthor( );
}

trace(asGuide.toString( )); // Displays: "A book about ActionScript, by Colin
      // But if we remove Book.prototype.toString,
      // Displays: [object Object]
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

```
// which is the return of Object.prototype.toString( ).
```

Now recall the syntax used to assign the superclass of a class:

```
// Make Book a subclass of Product
Book.prototype = new Product( );
```

Let's walk through how this statement establishes inheritance between *Book* and *Product* . First, we know that all *Book* instar all the properties defined on `Book.prototype` . We also know that, by default, `Book.prototype` is a generic instan *Object* class. Assigning *Product* as the superclass for *Book* , using this approach, replaces the generic instance in `Book.prototype` with a completely new instance of *Product* . Hence, all properties defined by *Product* become defined `Book.prototype` and therefore become available to all *Book* instances. With a *Product* instance stored in `Book.prot` , the prototype chain for our `asGuide` *Book* instance becomes:

```
Object.prototype      // If not found here, return undefined
Product.prototype     // If not found here, look on Object.prototype
Book.prototype        // If not found here, look on Product.prototype
asGuide               // If not found here, look on Book.prototype
                      // INTERPRETER STARTS LOOKING FOR PROPERTIES HERE
```

The prototype chain can include any number of levels. The *Product* class can itself have a superclass that hands down propert of its subclasses.

If you're dying to know whether an object defines its own property or inherits a property of the same name via the prototype c you can use the *Object.hasOwnProperty( )* method, described in the *Language Reference* . For example:

```
trace(asGuide.hasOwnProperty("author"));     // true
trace(asGuide.hasOwnProperty("getAuthor"));  // false
                                             // (because getAuthor is defined
                                             // on Book.prototype, not asGuide)
```

Conversely, if you want to access a property on the prototype chain that is overridden, you can use the `_ _proto_ _` prop (discussed next) to refer explicitly to the `prototype` property of an object's class. For example, the following code shows access `Book.prototype.description` through a particular *Book* instance, `asGuide` :

```
trace(asGuide._ _proto_ _.description);
```

12.7.1 The _ _proto_ _ Property

We saw earlier in this chapter that the *instanceof* operator tells us whether an instance belongs to a particular class. But how c interpreter know an individual object's class? Or, more specifically, how does the interpreter know which constructor's `prot` to access when searching for an object property? While that kind of internal information normally is hidden in object-orientec languages, ActionScript exposes it to the developer.

When any object is created, the interpreter automatically assigns it a special property called `_ _proto_ _` (note the *two* underscores on either side of the word `proto`). Then, into `_ _proto_ _` , the interpreter copies a reference to the object `prototype` property. For example, when we create an instance of *Book* called `asGuide` , `asGuide._ _proto_ _` `Book.prototype` :

```
var asGuide = new Book("Colin Moock", "A book about ActionScript");
trace(asGuide._ _proto_ _ =  = Book.prototype);  // Displays: true
```

Primarily, the `_ _proto_ _` property is used internally by the ActionScript interpreter. However, we can use it to determi

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

class of an object (as shown earlier in Example 12-5) or to access overridden properties. Although `_ _proto_ _` is writa should not be tampered with in most situations. However, soon we'll encounter a special case that uses `_ _proto_ _` as p alternative inheritance system.

12.7.2 The constructor Property

When a constructor's `prototype` object is created, the interpreter automatically assigns it a special property called `constructor` . The `constructor` property is a reference to the class constructor function on which the `prototype` is defined. For example, the following equality test checks whether `Book` and `Book.prototype.constructor` refe same function:

```
trace(Book.prototype.constructor =  = Book);  // Displays: true
```

Note that the `constructor` property contains a *reference* to a constructor function, not a string representing the function's The `constructor` can be used to invoke a constructor function on an object in order to reinitialize it. The following exam invokes the *Book* constructor on `asGuide` :

```
asGuide.constructor.call(asGuide,
                         "Colin Moock",
                         "A large book about ActionScript");
```

See *Function.call( )* in the *Language Reference* for details.

The `constructor` property can also be used to access static properties and methods from an instance. For example:

```
function Box ( ) {
  // Constructor code goes here
}
Box.numSides = 4;

var bigBox = new Box( );
trace(bigBox.constructor.numSides);  // Displays: 4
```

> In ECMA-262, JavaScript, and ActionScript, the `constructor` property is written to a class's prototype (as *`Class`* `.prototype.constructor`), where instances of the class can retrieve it via the prototype chain. However, ActionScript also adds a `constructor` property directly to instances (a *`instanceName`* `.constructor`) effectively obscuring *`Class`* `.prototype.constructor` . ActionScript's nonstandard implementation is considered a bug by Macromedia — the ECMA-262 standard requires that an instance retrieve its constructor via *`Class`* `.prototype.constructor` . In a future version of Flash, Macromedia will likely fix this bug (that is, in the future `constructor` will no longer be written directly to instances). We'll see how writing a `constructor` property to individual instances affects inheritance in the next section.

12.7.3 Advanced Issues with Standard SuperClass Assignment

As we've seen, the standard means for establishing a class's superclass is:

```
SubClass.prototype = new SuperClass( );
```

When using the *new* operator to establish inheritance, some developers have encountered minor problems, discussed in detail sidestep these problems, some developers use the following unofficial technique, so-called `_ _` *proto* `_ _` *-based* inheritanc

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

establish a class's superclass:

```
SubClass.prototype._ _proto_ _ = SuperClass.prototype;
```

Here's how `_ _proto_ _` -based inheritance works. We know that instances of *SubClass* have access to the properties and methods defined on `SubClass.prototype` . Further, we know that `SubClass.prototype` is itself an object that has the properties and methods of its own class via the `_ _proto_ _` property (remember the prototype chain?). Hence, `_ _proto_ _` -based inheritance says, "Give the `SubClass` .`prototype` object access to the properties and methods defined on `SuperClass.prototype` (via `SubClass.prototype._ _proto_ _`)."

Macromedia does not recommend using `_ _proto_ _` instead of the new operator to establish inheritance. However, establishing inheritance with `_ _proto_ _` is a relatively common advanced workaround that addresses the following issues.

12.7.3.1 Issue 1: Overwriting of prototype.constructor

Given a constructor function, *SubClass* , the property *`SubClass`* .`prototype.constructor` initially stores a reference to *SubClass* . What happens when you attempt to access the `constructor` property of an instance of *SubClass* as *`instanceName`* .`constructor` ? In ECMA-262, `constructor` is not defined on the instance, so the reference is resolved via the prototype chain, and the value of *`SubClass`* .`prototype.constructor` is returned. In ActionScript, however, `constructor` is defined on the instance itself, so *`instanceName`* .`constructor` is returned, and the interpreter doesn't look any further in the prototype chain. Let's see this in an example:

```
function Ball ( ) {
  // Class definition goes here
}
b = new Ball( );
trace(b.constructor);
```

In ECMA-262, `b.constructor` refers to the *Ball( )* constructor function as stored in `Ball.prototype.constructor` . In ActionScript for Flash Player 5 and Flash Player 6, `b.constructor` also refers to the *Ball( )* constructor function, but this time, the value is read from the `constructor` property attached directly to `b` .

For this simple example, `b.constructor` ultimately resolves to the same value in both ECMA-262 and ActionScript. So what's the matter that ECMA-262 reads the `constructor` property from the class's `prototype` , whereas ActionScript reads it from the separate instance? The answer lies in whether the two `constructor` properties are the same when `constructor` is accessed. In the simplest case, where an instance is created from a class, the discrepancy between ECMA-262 and ActionScript is indetectable because the instance's `constructor` value matches its class's `prototype.constructor` property. However, when a new superclass is established for a subclass using standard inheritance syntax (*SubClass.prototype = new SuperClass( );*), the subclass's previous `prototype` property is entirely replaced with a new *SuperClass* instance. Hence, any existing properties attached to `SubClass.prototype` , including `constructor` , are lost; conversely, all properties of the new *SuperClass* instance are added. Therefore, after an instance of *SuperClass* is assigned to *`SubClass`* .`prototype` , *`SubClass`* .`prototype` .`constructor` refers to the *SuperClass* constructor function, not the *SubClass* constructor as it once did.

The following code demonstrates this issue:

```
// Create SubClass
function SubClass ( ) {
   // Class definition goes here
}

// Check subclass constructor before inheritance is established
trace(SubClass.prototype.constructor =  = SubClass);    // Displays: true
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

```
// Create SuperClass
function SuperClass ( ) {
   // Class definition goes here
}

// Set inheritance using standard syntax (with the new operator)
SubClass.prototype = new SuperClass( );

// Check SubClass.prototype.constructor after inheritance is established
trace(SubClass.prototype.constructor =  = SubClass);    // Now displays: false
trace(SubClass.prototype.constructor =  = SuperClass);  // Now displays: true!
```

Note that *SubClass* .prototype.constructor matches the *SuperClass* class constructor function instead of the S class constructor function. What are the ramifications of this apparent discrepancy? And is ActionScript's departure from the 262 standard still moot?

Recall that in JavaScript and ECMA-262, *SubClass* instances retrieve the value of constructor from *SubClass* .prototype.constructor , so *SubClass* instances report the *SuperClass* 's constructor instead of their own *SubClass* constructor. This is the intentional, albeit confusing, implementation of the ECMA-262 standard and is not considered a bug.

Recall also that in Flash Player 5 and Flash Player 6, ActionScript deviates from the ECMA-262 standard and writes the constructor property directly to each instance. Hence, in Flash Player 5 and Flash Player 6, *anyInstance* .constructor always refers to the function that created *anyInstance* , in this case *SubClass* , even after superclass inheritance is established. For example:

```
subInstance = new SubClass();
trace(subInstance.constructor == SuperClass);  // false in Flash 5 and 6
trace(subInstance.constructor == SubClass);    // true in Flash 5 and 6
// Note that these equality tests always return true in Flash Player 5 due
// to an unrelated bug in the == operator (which was fixed in Flash Player 6).
```

Because it is obscured by subInstance.constructor , ActionScript programmers wouldn't ordinarily notice the mis *SubClass* .prototype.constructor property at all. Therefore the issue is somewhat moot in ActionScript for Fl Player 5 and 6. However, you should not rely on the constructor property that Flash 5 and 6 attach directly to instance because it is expected to disappear from future versions of the Flash Player.

Some readers might be tempted to "fix" the quirky ECMA-262 implementation by reassigning *SubClass* .prototype.constructor so that it refers to the original *SubClass* constructor function, as follows:

```
SubClass.prototype.constructor = SubClass;
```

However, Macromedia recommends against directly modifying *SubClass* .prototype.constructor , because do alters the intended (albeit, perhaps unexpected) ECMA-compliant value of the property. If you need an accurate and consister reference to an instance's constructor function, consider adding a custom property that stores a reference to the constructor (ac property after superclass assignment):

```
// Assign SuperClass
SubClass.prototype = new SuperClass( );
// Create custom constructorFunc property
SubClass.prototype.constructorFunc = SubClass;
```

If you are using the constructor property merely to determine whether an object belongs to a given class, use the *instan*

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

operator instead. The *instanceof* operator returns `true` when the specified instance belongs to the specified class or any of it subclasses. The *instanceof* operator works as expected, regardless of whether inheritance is establish with `_ _proto_ _` o the *new* operator, as shown here:

```
SubClass.prototype = new SuperClass();
var subInstance = new SubClass();
trace(subInstance instanceof SuperClass);  // Displays: true
trace(subInstance instanceof SubClass);    // Displays: true
```

12.7.3.2 Issue 2: Undesirable SuperClass constructor invocation

When the following statement is executed, the *SuperClass* function executes:

```
SubClass.prototype = new SuperClass( );
```

This extraneous function execution needlessly consumes resources and might perform unwanted tasks in the *SuperClass* cons (such as attaching a movie clip or incrementing an instance counter). To prevent unwanted code execution, use a parameter th indicates if the constructor is being called merely to establish inheritance. For example:

```
function SuperClass ( ) {
  // If the first argument is an object with a noConstruct property set to true
  if (arguments[0].noConstruct =  =  = true) {
    // ...abort construction
    return;
  }

  // Do regular construction
}
SubClass.prototype = new SuperClass( {noConstruct:true} );
```

In some cases, it may be satisfactory to simply check the number of arguments that were passed to the *SuperClass* constructor function. Here, the function expects two arguments; it exits if no arguments are passed in, as is typical when using *new* to esta inheritance:

```
function SuperClass (param1, param2) {
  // If no arguments were passed...
  if (arguments.length =  = 0) {
    // ...abort construction
    return;
  }

  // Do regular construction
}
SubClass.prototype = new SuperClass( );
```

In the alternative style of inheritance, based on `_ _proto_ _`:

```
SubClass.prototype._ _proto_ _ = SuperClass.prototype;
```

the `SuperClass.prototype` object is assigned directly to `SubClass.prototype._ _proto_ _`, and the *Su* constructor does not execute.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

12.7.3.3 Issue 3: Undefined properties appear in SubClass.prototype

A superclass constructor normally assigns properties to the instances it creates. In this example, the constructor *SuperClass* as instances a property, `p` :

```
function SuperClass (p) {
  this.p = p;
}
```

When inheritance is established with the standard *new* syntax, an instance of *SuperClass* is assigned to `SubClass.proto` and the property `p` is created on `SubClass.prototype` with a value of `undefined` (because a value for `p` was not p the *SuperClass* constructor). Hence, the `undefined` property `p` appears in *for-in* loops executed on all *SubClass* instances. example:

```
// SubClass constructor
function SubClass (p) {
  // Call superclass constructor
  super(p);
}

// SuperClass constructor
function SuperClass (p) {
  this.p = p;
}

// Establish inheritance
SubClass.prototype = new SuperClass( );

// Create an instance of SubClass with no value for p
subInstance = new SubClass( );

// Check subInstance's properties. The property p shows up
// even though it was not assigned a value when constructed.
for (var prop in subInstance) {
  // Displays: Property p is: undefined
  trace("Property " + prop + " is: " + typeof subInstance[prop]);
}
```

Though bothersome to some developers, the appearance of `p` on all instances of *SubClass* is normal and expected behavior. V class or superclass defines a property, that property is expected to be accessible on all instances, whether or not it contains use However, if preferred, we can prevent `undefined` properties from being assigned to instances as follows:

```
function SuperClass (p) {
  // Create p only if it is not undefined
  if (p != undefined) this.p = p;
}
```

That's not necessarily good OOP practice, but it does the job of hiding `p` when it has no value (again, as would be typical if th constructor was called to due a superclass assignment).

12.7.3.4 A custom setSuperClass( ) function

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

If you decide to use `_ _proto_ _` to establish inheritance (despite Macromedia's recommendations against it), consider c custom function that hides the slightly obtuse `_ _proto_ _` syntax. Example 12-6 shows a custom *setSuperClass( )* funct assigns a superclass to a subclass (i.e., establishes inheritance) using `_ _proto_ _` .

Example 12-6. A setSuperClass( ) function

```
// Add setSuperClass( ) to Function.prototype so it's
// available to all function objects
Function.prototype.setSuperClass = function (theSuperClass) {
  // Establish the super/sub class relationship.
  this.prototype._ _proto_ _ = theSuperClass.prototype;
}

// Create a superclass
function Circle (radius) {
  this.r = radius;
}

// Create a subclass
function Ball (radius) {
  // Can't use super when establishing inheritance with _ _proto_ _
  this.base = Circle;
  this.base(radius);
  delete this.base;
}

// Now establish inheritance with our custom setSuperClass( ) function
Ball.setSuperClass(Circle);

// Create an instance of Ball
var theBall = new Ball(15);

// Check the property assigned by the Circle class
trace(theBall.r);  // Displays: 15
```

Note that we might have naturally used the name "extends" instead of "setSuperClass" (following Java terminology), but "ext reserved as a potential keyword.

A word of warning to `_ _proto_ _` users: when inheritance is established via `prototype._ _proto_ _` , Flash P *super* operator cannot be used to invoke a superclass constructor. The superclass constructor must be invoked with the follow style syntax if inheritance was established using `_ _proto_ _` :

```
function SubClass (someArg) {
  this.base = SuperClass;
  this.base(someArg);
  delete this.base;
}
```

However, after establishing inheritance via `_ _proto_ _` , the *super* operator can still be used to invoke a superclass's me For example:

```
SubClass.prototype.hello = function ( ) {
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

```
  super.hello( );  // Works even when inheritance is
                   // established through prototype._ _proto_ _
}
```

12.7.4 Is There a Test?

While the preceding information about `prototype` , `constructor` , and `_ _proto_ _` is perhaps interesting to programming language enthusiasts, one might still wonder, must I really know it all? Does it change my use of classes and ob

Quite frankly, no, you should not have to know this information. The inner workings of a language's OOP implementation are typically the concern of the language's users. It's not normal for users to concoct their own OOP system, nor should developin OOP system be the user's responsibility. However, ECMAScript (the specification upon which ActionScript and JavaScript a based) was originally designed as a lightweight scripting language. The current demands put upon it have in some ways outst its expected purpose. Hence, ECMAScript developers have been forced to take matters into their own hands, attempting to fix problems inherent in the language implementation, in order to achieve their goals. Meanwhile, work is underway to complete revise the inheritance system in ECMAScript 4 and JavaScript 2.0. See:

http://www.mozilla.org/js/language/js20
http://www.mozilla.org/js/language/es4

At the time of this writing, the ECMAScript 4 specification was very much under development, so Macromedia's position on future of ECMAScript remains to be seen.

In any case, the only time you should concern yourself with `prototype` and `_ _proto_ _` is when you hit a real limita your work. Unless the standard inheritance system in ActionScript causes problems for your programs, you should not bother `_proto_ _` ; however, if you encounter problems caused by standard inheritance, investigate alternatives. Luckily, other developers have paved some of the way for you. See Dave Yang's information for one perspective:

http://www.quantumwave.com/flash/inheritance.html

Branden Hall and Robert Penner have also written about inheritance issues (see their books, listed in Appendix A). For the m however, you can simply use the standard means of establishing inheritance (assigning a new instance of a superclass to the s `prototype` property) without worrying about the technicalities of the prototype system.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Chapter 12. Objects and Classes

12.8 Built-in ActionScript Classes and Objects

We've really come a long way since Chapter 1. Our first exposure to ActionScript was a simple perusal of the items in the Actions panel. Since then, we've studied data and expressions, operators, statements, and functions; and now we've explored the concept of classes and objects. It's time to put the final brush strokes on the picture we've been painting of the ActionScript language.

ActionScript comes with a variety of syntactic tools: *expressions* contain data; *operators* manipulate data; *statements* give instructions; and *functions* group instructions into portable commands. They constitute ActionScript's grammar, but until now, our vocabulary has been very limited. The built-in classes and objects of ActionScript fill that void and give us lots of interesting things to talk about.

12.8.1 Built-in Classes

Just as we define our own classes to describe and manipulate objects created according to our specifications, ActionScript defines its own classes of data. A variety of prefabricated classes, including the *Object* class, are built right into the ActionScript language. The built-in classes can control the physical environment of a Flash movie.

For example, one of the built-in classes, the *Color* class, defines methods that can detect or set the color of a movie clip. To use these methods, we first create a *Color* object using the *Color( )* class constructor, as follows:

```
clipColor = new Color(target);
```

where `clipColor` is a variable, array element, or object property that stores our *Color* object. The *Color( )* constructor takes one argument, `target`, which specifies the name of the movie clip whose color we want to set or examine.

So, suppose we have a movie clip named `square`, and we want to change its color. We create a new *Color* object like this:

```
squareColor = new Color(square);
```

Then, to set our `square` clip's color, we invoke one of the *Color* methods on our `squareColor` object:

```
squareColor.setRGB(0x999999);
```

The *setRGB( )* method sets the RGB color value of our *Color* object's `target`, which in this case is `square`. So, the previous method invocation sets the color of `square` to gray.

Because the *Color* object is a built-in class, it can set the color of a movie clip on screen. Other classes give us control over sounds, text fields, buttons, the date, and XML documents. You'll learn all about the built-in classes in the ActionScript Language Reference.

12.8.2 Built-in Objects

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Like built-in classes, built-in objects let us control the movie environment. The *Key* object, for example, defines a series of properties and methods that tell us about the state of a computer's keyboard. To use these properties and methods, we don't instantiate a *Key* object, we simply use the *Key* object directly. Built-in objects are made automatically by the interpreter when the Flash Player starts, and they are globally available throughout a movie.

Here, for example, we display the keycode of the currently depressed key by invoking the *Key* object's *getCode( )* method:

```
trace(Key.getCode());
```

And here, we check whether the spacebar is depressed, using the *isDown( )* method and supplying the spacebar's keycode as an argument:

```
trace(Key.isDown(Key.SPACE));
```

In the ActionScript Language Reference, you'll learn about the other built-in objects.

12.8.3 Learning the Ropes

Learning to write valid ActionScript code is only half the job of learning to program in Flash. The other half comes with learning the available built-in classes and objects and their many wonderful properties and methods. We'll undertake that task in the ActionScript Language Reference. However, it's not necessary to study all the classes and objects in one sitting. Learn about movie clips to start, and branch out as you need to. Over time, you'll come to know what's essential to get your particular job done. What's important is that you understand the general structure of object-oriented programming. Once you know the rules of the system, learning a new object or a new class is a simple matter of looking up its method and property names.

12.8.4 Adding Custom Methods and Properties to Built-in Classes and Objects

Though the built-in classes and objects are predetermined by ActionScript, we can still give them new abilities by adding custom methods and properties to them exactly as we do with own custom classes and objects. Entirely new classes can also be derived from built-in classes.

The following code adds a *randomInt( )* method to the static *Math* object:

```
// Returns a number in the range minVal to maxVal, inclusive
Math.randomInt = function (minVal, maxVal) {
  return minVal + Math.floor(Math.random() * (maxVal + 1 - minVal));
};
```

The following code adds a *scrollToBottom( )* method to the *TextField* class:

```
TextField.prototype.scrollToBottom = function ( ) {
  this.scroll = this.maxscroll;
};
```

The following code creates a subclass of the *XML* class:

```
function CustomXML (src) {
  super(src);
}
CustomXML.prototype = new XML( );
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Some built-in classes actually require the execution of their constructor function as part of the setup for new objects. When in doubt, always invoke the superclass constructor as shown in our previous examples. Note that not all built-in classes can be subclassed, because some built-in classes (e.g., *TextField*, *Button*) are tied to physical objects on screen. See each class entry in the ActionScript Language Reference for details.

Many, many more examples of extending the built-in classes and objects can be found on the developer community site [PROTO]type:

http://www.layer51.com/proto/

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Chapter 12. Objects and Classes

12.9 OOP Quick Reference

We've covered a lot of material in this chapter. For review and reference, here's a quick-and-dirty overview of OOP technique ActionScript. At the end, you'll find a template for creating a new OOP application.

To create a class:

```
function ClassName (param1, param2) {
  // ...constructor code goes here...
}
```

To create instance properties for a class:

```
function ClassName (param1, param2) {
  // Instance properties
  this.propName1 = param1;
  this.propName2 = param2;
}
```

Best practice is to create all properties used in a class's methods in the class constructor, even if the properties initially have n value:

```
function ClassName (param1, param2) {
  // Declare all instance properties
  this.propName1 = param1;
  this.propName2 = param2;
  this.propName3 = null;  // Used by method1( );
  this.propName4 = null;  // Used by method2( );
}
```

After creating properties, initialize the object:

```
function ClassName (param1, param2) {
  // Declare all instance properties
  this.propName1 = param1;
  this.propName2 = param2;

  // Initialize object.
  // ...initialization code goes here...
}
```

Instead of putting the initialization code in the constructor, consider using an *init( )* method, which can be called to reset the object to its default state:

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

```
function ClassName (param1, param2) {
  // Declare all instance properties
  this.propName1 = null;
  this.propName2 = null;

  // Initialize object
  this.init.apply(this, arguments);
}

ClassName.prototype.init = function (param1, param2) {
  this.prop1 = param1;
  this.prop2 = param2;
};
```

To create an instance method:

```
ClassName.prototype.methodName = function ( ) {
  // ...method body goes here...
};
```

To refer to an instance within a method of the instance, use the `this` keyword:

```
ClassName.prototype.methodName = function ( ) {
  this.propName1 = value;
  this.doSomething( );
};
```

To create a static property, attach it to the class constructor function:

```
ClassName.staticPropName = value;
```

To create a static method, attach it to the class constructor function:

```
ClassName.staticMethodName = function ( ) {
  // ...method body goes here...
};
```

To create a subclass, set the `prototype` property of the class constructor function:

```
function SubClassName (param1, param2) {
  // ...constructor code goes here...
}
// DO THIS BEFORE DEFINING SubClassName's METHODS!
SubClassName.prototype = new SuperClassName( );
```

12.9.1 Some OOP Tips

We've covered a lot of conceptual ground, so let's synthesize some of these ideas to help you put OOP into practice.

- Define everything in a class. All variables and functions should be properties and methods of a class. This ensures that properties and methods are grouped logically and are always qualified by some object (which helps avoid scope proble

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

- For the sake of organization and reuse across projects, store each class in a single, separate *.as* text file.
- Define all classes on `_global` .
- Use simulated namespaces. Create an object on `_global` that will hold all your classes, as shown in Section 12.9.2.
- Create a single *main( )* function that starts your application rolling.
- Create all instance methods on your constructor's `prototype` object, not in the constructor.
- Always document your classes, properties, and methods. For an example, see Section 12.9.2. In your documentation, specify which methods and properties are intended for public use and which are intended for private, internal use by the class.

12.9.2 OOP Application Template

Sometimes ActionScript is so flexible that it becomes slippery. There are so many ways to create a legal object-oriented application that finding a standard approach can become an arduous task. The following sample template for an object-oriente application attempts to provide some initial direction. You should adapt it to your needs. An electronic version of the templat available at the online Code Depot.

12.9.2.1 Main application template

The following code should be attached to the main timeline of *yourApp.fla* , where *yourApp.fla* is your application's main Fla source file. The frame to which you attach the code should be preloaded (see *`MovieClip` `._framesloaded`* in the ActionScript Language Reference for preloading information).

```
// =  =  =  =  =  =  =  =  =  =  =  =  =  =  =  =  =  =  =  =  =  =  =
// LOAD CLASSES
// =  =  =  =  =  =  =  =  =  =  =  =  =  =  =  =  =  =  =  =  =  =  =

// Create namespace in which to store classes
if (_global.com =  = undefined) {
  _global.com = new Object( );
}
if (_global.com.yourDomain =  = undefined) {
  _global.com.yourDomain = new Object( );
}
// Load external .as files, one per class
#include "YourClass.as"
#include "YourSubClass.as"

// =  =  =  =  =  =  =  =  =  =  =  =  =  =  =  =  =  =  =  =  =  =  =
// START APPLICATION
// =  =  =  =  =  =  =  =  =  =  =  =  =  =  =  =  =  =  =  =  =  =  =
main( );

// Function: main( )
//     Desc: The single entry point for your application
function main ( ) {
  // Create and use instances of your classes
  var yourInstance = new com.yourDomain.YourClass(arg1, arg2);
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

```
  var yourSubInstance = new com.yourDomain.YourSubClass(arg1, arg2, arg3, arg4
  yourInstance.yourMethod( );
  yourSubInstance.yourSubMethod( );
  yourSubInstance.yourMethod( );
}
```

12.9.2.2 Class template

The following code should be saved in a file named *YourClass.as* . In this example, *YourClass.as* and *yourApp.fla* reside in t same directory.

```
/*
 * YourClass Class
 *   Version: 1.0.0
 *   Desc: Your description goes here.
 *
 * Constructor Params:
 *   param1              -Short description
 *   param2              -Short description
 *
 * Methods:
 *   yourMethod( )       -Short description
 *
 * Static Properties
 *   staticProp1         -Short description
 *   staticProp2         -Short description
 *
 * Static Methods
 *   staticMeth( )       -Short description
 */

/*
 * Class Constructor
 */
_global.com.yourDomain.YourClass = function (param1, param2) {
  // Create instance properties
  this.prop1 = param1;  // Short description. Type: state datatype
  this.prop2 = param2;  // Short description. Type: state datatype
  this.prop3 = null;    // Short description. Type: state datatype

  // Initialize object
  // ...initialization code goes here...
  // Consider packaging init code into an init( ) method
};

/*
 * Static Properties
 */
com.yourDomain.YourClass.staticProp1 = value;
com.yourDomain.YourClass.staticProp2 = value;

/*
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

```
 * Static Methods
 */
com.yourDomain.YourClass.staticMeth = function ( ) {

};

/*
 * Instance Methods
 */

/*
 * Method: YourClass.yourMethod( )
 *   Desc: Short description.
 *
 * Params:
 *   param1              -Short description
 */
com.yourDomain.YourClass.prototype.yourMethod = function (param1) {
  // Method body...
  trace("yourMethod invoked");
};
```

12.9.2.3 SubClass template

The following code should be saved in a file named *YourSubClass.as* . In this example, *YourSubClass.as* and *yourApp.fla* res in the same directory.

```
/*
 * YourSubClass Class. Extends YourClass.
 *   Version: 1.0.0
 *   Desc: Your description goes here.
 *
 * Constructor Params:
 *   param1              -Superclass parameter
 *   param2              -Superclass parameter
 *   param3              -Short description
 *   param4              -Short description
 *
 * Methods:
 *   yourSubMethod( )  -Short description
 */

/*
 * Class Constructor
 */
_global.com.yourDomain.YourSubClass = function (param1, param2, param3, param4
  // Invoke superclass constructor.
  super(param1, param2);

  // Create instance properties.
  this.subProp1 = param3;  // Short description. Type: state datatype
  this.subProp2 = param4;  // Short description. Type: state datatype
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

```
  this.subProp3 = null;    // Short description. Type: state datatype

  // Initialize object.
  // ...initialization code goes here...
  // Consider packaging init code into an init( ) method
};

/*
 * Establish Inheritance. MUST COME BEFORE METHODS!
 */
com.yourDomain.YourSubClass.prototype = new com.yourDomain.YourClass( );

/*
 * Instance Methods
 */

/*
 * Method: YourSubClass.yourSubMethod( )
 *   Desc: Short description.
 *
 * Params:
 *   param1              -Short description
 */
com.yourDomain.YourSubClass.prototype.yourSubMethod = function (param1) {
  // Method body...
  trace("yourSubMethod invoked");
};
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Chapter 12. Objects and Classes

12.10 Further Topics

The scope of this book precludes an in-depth discussion of the following topics. You are encouraged to pursue them using the external sources listed.

12.10.1 Custom Events

You may have noticed that you have learned how to create methods and properties, but not event handlers or listener events. ActionScript does not provide a documented means of defining custom events for a class. However, the undocumented *ASBroadcaster* provides internal support for custom events. For details, see:

http://chattyfig.figleaf.com/flashcoders-wiki/index.php?AsBroadcaster

Branden Hall has also developed a custom event engine known as FLEM. See:

http://chattyfig.figleaf.com/flem/
http://chattyfig.figleaf.com/flem/extend.php

12.10.2 Design Patterns and UML

Once you've learned the syntax and methodology of object-oriented programming, you'll eventually realize that the challenging part is not the technical implementation but the design of the classes used to represent any given application. In the OOP world, consistent designs for similar problems have arisen and been dubbed "design patterns." Though there is little design pattern documentation available that specifically addresses ActionScript, design patterns are intended to be applied to any OOP-based language. For the adventurous ActionScripter willing to learn a little Java and/or C++, the following books are recommended:

Design Patterns Explained: A New Perspective on Object-Oriented Design, Shalloway and Trott, Addison Wesley, 2001.
Design Patterns: Elements of Reusable Object-Oriented Software, Gamma et al, Addison Wesley, 1995.

Typically, design patterns are charted with a special diagram language known as UML (Unified Modeling Language). UML is a standard for depicting class hierarchies visually. For information, see:

http://www.omg.org/gettingstarted/what_is_uml.htm
http://www.devx.com/uml/
UML Distilled: A Brief Guide to the Standard Object Modeling Language, Fowler and Scott, Addison Wesley, 1999.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Chapter 12. Objects and Classes

12.11 Simulating Namespaces

A *namespace* is a unique area of a program within which all identifiers are guaranteed to be unique. It lets a program use, say, both Company X's and Company Y's unique *Ball* classes without conflicts (i.e., without one class declaration overwriting the other). ActionScript in Flash MX does not have formal support for namespaces, but we can simulate them by creating a unique global object and defining all of a program's identifiers on it (as we saw in the earlier Section 12.9.2). Typically, the unique global object is named after a domain name, because domains are guaranteed to be unique. For example, to create a namespace for my classes I use the following code (my site's domain is *moock.org*):

```
if (_global.org =  = undefined) {
  _global.org = new Object( );
}
if (_global.org.moock =  = undefined) {
  _global.org.moock = new Object( );
}
```

This code appears in all my classes (each class has the responsibility of creating the namespace if it doesn't already exist.) Here's a convenient function, *AsSetupPackage( )*, that you can use to create a namespace for your own classes:

```
_global.AsSetupPackage = function (path) {
  var a = path.split('.');
  var o = _global;
  for (var i = 0; i < a.length; i++) {
    var name = a[i];
    if (o[name] =  = undefined) {
      o[name] = new Object( );
    }
    o = o[name];
  }
}
```

Within each class, you can use *AsSetupPackage( )* to create the class's namespace, as follows:

```
AsSetupPackage("com.yourdomain");
```

For example, I use:

```
AsSetupPackage("org.moock");
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Chapter 12. Objects and Classes

12.12 The Multiple-Choice Quiz, OOP Style

Now that you understand the principles of OOP, let's return to our multiple-choice quiz example to see them in action. The source code for our OOP quiz is too long to present here, but it can be downloaded from the Code Depot. The important part of the application is its object design, listed next. Even without seeing the code in the quiz, the following tables—which describe the classes, methods, properties, and events in the quiz—should give you a complete picture of how the application works. In fact, using the following structure, you may want to try to build the quiz yourself before you download its source code. Note that some properties are not listed in the tables, because they are used by a class's methods only, and they are therefore considered internal ("private") to the class.

Our quiz comprises three classes, shown in Table 12-2. The entire quiz is generated from ActionScript—there's not a single asset in the movie's Library. All buttons are drawn with the Drawing API, and all text is created with *MovieClip.createTextField( )*, giving the quiz a flexible layout that adapts to the current Player size, question length, and answer length. Navigation buttons—Next, Back, Grade, and Try Again—let the user proceed nonlinearly through the quiz, change an existing answer, be graded, or retry the quiz.

Table 12-2. Multiple-choice quiz classes

Class name	Class description
Quiz	Manages the data (questions and answers) in the quiz Handles loading questions from an external XML file Starts the quiz and tracks the current question Grades the user
QuizGUI	Displays the quiz on screen and receives user input
Question	Manages the data of an individual question and its answer

The *Quiz* class defines the methods and properties listed in Table 12-3.

Table 12-3. The Quiz class

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Name	Type	Description
loadQuizData( )	Method	Loads an XML file and generates the quiz based on it
startNewQuiz( )	Method	Resets the quiz and tells the *QuizGUI* it's time to begin the quiz
setQuizGUI	Method	Stores a reference to a *QuizGUI* instance that will represent the quiz on screen
setTitle( )	Method	Stores the quiz title
getTitle( )	Method	Returns the quiz title
setDescription( )	Method	Stores a short text description for the quiz
getDescription( )	Method	Returns the quiz description
getScore( )	Method	Calculates and returns the user's current score, based on the questions answered so far
setCurrentQuestion( )	Method	Sets the question being answered
getCurrentQuestion( )	Method	Returns the index of the question currently being answered
getNumUserAnswers( )	Method	Returns the number of questions answered so far
setNumUserAnswers( )	Method	Sets the number of questions answered so far
setUserAnswer( )	Method	Stores the user's answer for a specific question
addQuestion( )	Method	Inserts a new *Question* object into the quiz
getQuestionAt( )	Method	Returns a specific *Question* object from the list of questions in the quiz
getNumQuestions( )	Method	Returns the number of questions in the quiz
isComplete( )	Method	Returns `true` if the quiz is considered finished; otherwise it returns `false`. In the simplest case, a quiz is complete when all the questions have been answered by the user.
`quizGUI`	Property	A reference to the *QuizGUI* instance that will display the quiz

The *QuizGUI* class defines the methods, properties, and events listed in Table 12-4. Note that the event handlers on the *QuizGUI* class should more appropriately be defined as *Quiz* listener events. However, because ActionScript does not provide a built-in means of creating listener events, we keep the code simple by placing the handlers directly on *QuizGUI*.

Table 12-4. The QuizGUI class

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Name	Type	Description
displaySplashScreen( )	Method	Displays a welcome screen for the quiz
displayQuestion( )	Method	Displays an entire question, including answer and navigation buttons
displayQuizEnd( )	Method	Displays a final status report for the quiz, showing the user's score
displayTitle( )	Method	Displays the quiz title on screen
createNavButton( )	Method	A convenience function that creates and returns a navigation button
clearScreen( )	Method	Removes all visual assets from the screen
onQuizStart( )	Event Handler	Invoked when a new quiz begins
onQuestionUpdate( )	Event Handler	Invoked when the content of a question changes, the user submits an answer, or the current question index changes
onQuizComplete( )	Event Handler	Invoked when the user has answered every question
`quizObj`	Property	A reference to the *Quiz* instance for which this *QuizGUI* is handling display
`quizClip`	Property	A reference to the movie clip in which the quiz display assets will be created

The *Question* class defines the methods listed in Table 12-5. The *Question* constructor can receive question data either as a series of values or as a single XML fragment.

Table 12-5. The Question class

Name	Type	Description
setQuestionText( )	Method	Assigns the text for the question
getQuestionText( )	Method	Returns the text for the question
setAnswers( )	Method	Assigns the multiple-choice answers, as an array
getAnswers( )	Method	Returns the multiple-choice answers, as an array
setCorrectAnswer( )	Method	Assigns the correct answer for the question, as an index in the answers array
getCorrectAnswer( )	Method	Returns the correct answer for the question, as an index in the answers array
setUserAnswer( )	Method	Assigns the user's answer for the question, as an index in the answers array. This can be set more than once.
getUserAnswer( )	Method	Returns the user's current answer for the question, as an index in the answers array

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Chapter 12. Objects and Classes

12.13 Onward!

Classes and inheritance open up a world of potential for sharing common information among many objects. Object-oriented programming is the basis of all code in ActionScript. Whether or not you use every aspect of classes and objects in your scripts, understanding the general concepts involved is an essential part of understanding the Flash programming environment. Confidence with object-oriented programming greatly influences confidence with ActionScript in general.

For further reading on object-oriented programming with ECMA-262-derived languages, see Netscape's documentation for JavaScript, *Details of the Object Model*:

http://developer.netscape.com/docs/manuals/js/core/jsguide/obj2.htm

David Flanagan's canonical text, *JavaScript: The Definitive Guide* (O'Reilly), also provides valuable information on OOP in JavaScript.

For a general introduction to OOP from a Java perspective, see Sun's *Object Oriented Programming Concepts* (from "The Java Tutorial"):

http://java.sun.com/docs/books/tutorial/java/concepts

For a complete online walkthrough of OOP in ActionScript, try Robin Debreuil's tutorial:

http://www.debreuil.com/docs/

And, for the curious, here's an extensive set of articles attempting to deflate the oft-hyped OOP system. Whether you agree or disagree with the author, you'll likely learn from the discussion.

http://www.geocities.com/tablizer/oopbad.htm

We've certainly come a long way, and I hope you're still as enthusiastic about the journey as I am. Next we'll explore the most important class of objects available in ActionScript, the MovieClip class.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Part I: ActionScript Fundamentals

Chapter 13. Movie Clips

Every Flash document contains a Stage—on which we place shapes, text, and other visual elements—and a main timeline, through which we define changes to the Stage's contents over time. The Stage (i.e., the *main movie*) can contain independent submovies, called *movie clips* (or *clips* for short). Each movie clip has its own independent timeline and *canvas* (the Stage is the canvas of the main movie) and can even contain other movie clips. A clip contained within another clip is called a *nested clip.* A clip that contains another clip is referred to as the nested clip's *host clip* or *parent clip.*

A single Flash document can contain a hierarchy of interrelated movie clips. For example, the main movie may contain a mountainous landscape. A separate movie clip containing an animated character can be moved across the landscape to give the illusion that the character is walking. Another movie clip inside the character clip can be used to animate the character's blinking eyes independently. When the independent elements in the cartoon character are played back together, they appear as a single piece of content. Furthermore, each component can react intelligently to the others; we can tell the eyes to blink when the character stops moving or tell the legs to walk when the character starts moving.

ActionScript offers detailed control over movie clips; we can play a clip, stop it, move its playhead within its timeline, programmatically set its properties (such as its size, rotation, transparency level, and position on the Stage) and manipulate it as a true programming object. As a formal component of the ActionScript language, movie clips can be thought of as the raw material used to produce programmatically generated content in Flash. For example, a movie clip can serve as a ball or a paddle in a pong game, as an order form in a catalog web site, or simply as a container for background sounds in an animation.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Chapter 13. Movie Clips

13.1 The "Objectness" of Movie Clips

As of Flash 5, movie clips can be manipulated like the objects we learned about in Chapter 12. We can retrieve and set the properties of a clip, and we can invoke built-in or custom methods on a clip. An operation performed on a clip often has a visible or audible result in the Flash Player.

Movie clips are not truly a type of object, but they are object-like; although we can neither create movie clips via a class constructor nor use an object literal to instantiate a movie clip, we can instantiate movie clips using *attachMovie( )* , *duplicateMovieClip( )* and *createEmptyMovieClip( )* (introduced in Flash Player 6). So what, then, are movie clips, if not objects? They are members of their very own object-like datatype, called *movieclip* (we can prove it by executing *typeof* on a movie clip, which returns the string "movieclip"). The main difference between movie clips and other objects is how they are allocated (created) and deallocated (disposed of, or freed). For details, see:

http://www.moock.org/asdg/technotes/movieclipDatatype

Despite this technicality, however, we nearly always treat movie clips exactly as we treat objects.

So how does the "objectness" of movie clips affect our use of them in ActionScript? Most notably, it allows us to call methods on clips and examine their properties, just as we can for other objects. Movie clips can be controlled directly through built-in methods. For example:

```
eyes_mc.play( );
```

We can retrieve and set a movie clip's properties using the dot operator, just as we access the properties of any object:

```
ball_mc._xscale = 90;
var radius = ball_mc._width / 2;
```

A variable in a movie clip is simply a property of that clip, and we can use the dot operator to set and retrieve variable values:

```
theClip_mc.someVariable = 14;
x = theClip_mc.someVariable;
```

Nested movie clips can be treated as object properties of their parent movie clips. We therefore use the dot operator to access nested clips:

```
clipA.clipB.clipC.play();
```

and we use the reserved `_parent` property to refer to the clip containing the current clip:

```
_parent.clipC.play();
```

Treating clips as objects affords us all the luxuries of convenient syntax and flexible playback control. But our use of

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

clips as objects also lets us manage clips as data; we can store a movie clip reference in an array element or a variable, and we can even pass a clip reference to a function as an argument. Here, for example, is a function that moves a clip to a particular location on the screen:

```
function moveClipTo (clip, x, y) {
  clip._x = x;
  clip._y = y;
}
moveClipTo(ball_mc, 14, 399);
```

In this chapter, we'll cover the specifics of referencing, controlling, and manipulating movie clips as data objects.

13.1.1 The MovieClip Class

All individual movie clips are instances of the *MovieClip* class, which defines the properties, methods, and event handlers supported by movie clips. The *MovieClip* class can be manipulated like other built-in classes; we can overwrite its methods or add new methods to it using the techniques we studied in Chapter 12. For example, the following code adds a new method, *getArea( )*, to all movie clips:

```
MovieClip.prototype.getArea = function ( ) {
  return this._width * this._height;
};
```

As we'll see in Chapter 14, as of Flash MX, the *MovieClip* class can even be used as the superclass for new classes (i.e., new subclasses can be derived from the *MovieClip* class). For full coverage of every property, method, and event handler supported by the *MovieClip* class, see the *Language Reference*.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Chapter 13. Movie Clips

13.2 Types of Movie Clips

Not all movie clips are created equal. In fact, there are three distinct types of clip available in Flash:

- Main movies
- Regular movie clips
- Components (formerly known as *Smart Clips* in Flash 5)

In addition to these three official varieties, we can distinguish four subcategories of regular movie clips:

- Process clips
- Script clips
- Linked clips
- Seed clips

While these unofficial subcategories are not formal terms used in ActionScript, they provide a useful way to think about programming with movie clips. Let's take a closer look at each movie clip type.

13.2.1 Main Movies

The *main movie* of a Flash document is the main timeline and Stage present in every *.swf* document. The main movie is the foundation for all the content in the document, including all other movie clips. We sometimes refer to the main movie as the *main timeline*, the *main movie timeline*, the *main Stage*, or simply the *root*.

Note that while each *.swf* file contains only one main movie, more than one *.swf* file can reside in the Flash Player at once—we can load multiple *.swf* documents (and therefore multiple main movies) onto a stack of *levels* via the *loadMovie( )* and *unloadMovie( )* functions, which we'll study later.

Main movies can be manipulated in much the same way as regular movie clips, with the following exceptions:

- A main movie cannot be removed from a *.swf* file (although a *.swf* file, itself, can be removed from the Flash Player).
- The following movie clip methods do not work when invoked on a main movie: *duplicateMovieClip( )*, *removeMovieClip( )*, and *swapDepths( )*.
- The following properties are not supported by main movies: `enabled`, `focusEnabled`, `hitArea`,

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

`_name`, `_parent`, `tabChildren`, `tabEnabled`, `tabIndex`, `trackAsMenu`, and `useHandCursor`.

- In Flash 5, event handlers cannot be attached to the main movie. As of Flash Player 6, the following event handler properties can be assigned to the main movie: *onData( )*, *onEnterFrame( )*, *onKeyDown( )*, *onKeyUp( )*, *onMouseDown( )*, *onMouseMove( )*, and *onMouseUp( )*.
- Main movies cannot receive keyboard input focus.
- Main movies can be referenced through the built-in global `_root` and `_level`*n* properties.

13.2.2 Regular Movie Clips

Regular movie clips are the most common and fundamental content containers; they hold visual elements and sounds, and they can even react to user input and movie playback through event handlers. For JavaScript programmers who are used to working with DHTML, it may be helpful to think of the main movie as analogous to an HTML document object and regular movie clips as analogous to that document's layer objects.

13.2.3 Components

An improvement over Flash 5 Smart Clips, a Flash MX *component* is a movie clip that includes a graphical user interface used to customize the clip's properties in the authoring tool. Components typically are developed by advanced programmers to distribute as self-contained program modules, such as a pull-down menu or slider bar. Components also provide an easy way for less-experienced Flash authors to customize a movie clip's behavior without knowing how the code of the clip works. We'll cover components in detail in Chapter 14 and Chapter 16. Flash MX comes with a collection of ready-made interface components known as the Flash UI Components. To access the Flash UI Components, choose Window Components.

13.2.4 Process Clips

A *process clip* is a movie clip used not for content but simply to execute a block of code repeatedly. Process clips can be built with an *onEnterFrame( )* event handler or with a timeline loop, as we saw in Chapter 8 under Section 8.7.

As of Flash MX, the functionality of process clips can be partially replaced by the *setInterval( )* function, which executes a function or method periodically. Timed code should be implemented with *setInterval( )*, whereas code synched specifically with the frame rate should be implemented with a process clip. See the ActionScript Language Reference for a discussion of *setInterval( )*.

13.2.5 Script Clips

Like a process clip, a *script clip* is an empty movie clip used not for content but for tracking some variable, defining some class, or simply executing some arbitrary script. For example, in Flash 5, we can use a script clip to hold event handlers that detect keypresses or mouse events. In Flash MX, we can use a script clip to create a so-called "code only" component.

13.2.6 Linked Clips

A *linked clip* is a movie clip that either exports from or imports into the Library of a movie. Export and import settings are available through every movie clip's Linkage option, found in the Library. We most often use linked clips when dynamically generating an instance of a clip directly from a Library symbol using the *attachMovie( )* method, as

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

we'll see later.

13.2.7 Seed Clips

Before the *attachMovie( )* method was introduced in Flash 5, we used the *duplicateMovieClip( )* function to create new movie clips based on some existing clip, called a *seed clip*. A *seed clip* is a movie clip that resides on stage solely for the purpose of being copied via *duplicateMovieClip( )*. With the introduction of *attachMovie( )*, which instantiates clips from the Library, the need for on-stage seed clips has diminished. However, we still use seed clips and *duplicateMovieClip( )* when we wish to retain a clip's transformations and *onClipEvent( )* handlers in the process of copying it.

In a movie that makes heavy use of *duplicateMovieClip( )* to dynamically generate content, it's common to see a row of seed clips on the outskirts of the movie canvas. The seed clips are used only to derive duplicate clips and are, therefore, kept off stage.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Chapter 13. Movie Clips

13.3 Creating Movie Clips

We usually treat movie clips just as we treat objects—we set their properties with the dot operator; we invoke their methods v the function-call operator (parentheses); and we store them in variables, array elements, and object properties. We do not, however, create movie clips in the same way we create objects. We cannot literally describe a movie clip in our code as we m describe an object with an object literal. And we cannot generate a movie clip with the *new* operator:

```
myClip = new MovieClip( );  // Nice try buddy, but it won't work
```

Although Flash Player 6 supports the *new MovieClip( )* command, that command establishes a *MovieClip* subclass; it *cannot used to create a new movie clip instance in a movie* . Instead, we normally create movie clips directly in the authoring tool, by hand. Once a clip is created, we can use commands such as *duplicateMovieClip( )* and *attachMovie( )* to make new, independ duplicates of it. As of Flash Player 6, we can also create a completely new movie clip at runtime with the *createEmptyMovieClip( )* method.

13.3.1 Movie Clip Symbols and Instances

Just as all object instances are based on a class, all movie clip instances are based on a template movie clip, called a *symbol* (sometimes called a *definition*). A movie clip's symbol acts as a model for the clip's content and structure. With the exceptior clips created via *createEmptyMovieClip( )* , we generate a specific clip instance from a movie clip symbol stored in the Libra Using a symbol, we can both manually and programmatically create clips to be rendered in a movie.

A specific copy of a movie clip symbol is called an *instance* . Instances are the individual clip objects that can be manipulatec with ActionScript; a *symbol* is the mold from which instances of a specific movie clip are derived. Movie clip symbols are created in the Flash authoring tool. To make a new, blank symbol, we follow these steps:

1. Select Insert New Symbol. The Create New Symbol dialog box appears.
2. In the Name field, type an identifier for the symbol.
3. For Behavior, select the Movie Clip radio button.
4. Click OK.

Normally, the next step is to fill the symbol's canvas and timeline with the content of our movie clip. Once a symbol has been created, it resides in the Library, waiting for us to use it to instantiate a movie clip instance. However, it is also possible to convert a group of shapes and objects that already exist on stage into a movie clip symbol. To do so, we follow these steps:

1. Select the desired shapes and objects.
2. Select Insert Convert to Symbol.
3. In the Name field, type an identifier for the symbol.
4. For Behavior, select the Movie Clip radio button.
5. Click OK.

The shapes and objects we select to create the new movie clip symbol are replaced by an unnamed instance of that new clip. T corresponding movie clip symbol appears in the Library, ready to be used to create further instances.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

13.3.2 Creating Instances

There are four ways to create a new movie clip instance. Three of these are programmatic; the other is strictly manual and is undertaken in the Flash authoring tool. All but one method, *createEmptyMovieClip( )* , require an existing movie clip symbol from which to derive the new instance.

13.3.2.1 Manually creating instances

We can create movie clip instances manually using the Library in the Flash authoring environment. By physically dragging a movie clip symbol out of the Library and onto the Stage, we generate a new instance. An instance created in this way should l named manually via the Instance panel (Flash 5) or Property inspector (Flash MX). You'll learn more about instance names la in this chapter. Refer to "Using Symbols, Instances, and Library Assets" in the Macromedia Flash Help if you've never worke with movie clips in Flash.

13.3.2.2 Creating instances with duplicateMovieClip( )

Any instance that already resides on the Stage of a Flash movie can be duplicated with ActionScript. We can then treat that cc as a completely independent clip. Both manually created and programmatically created clip instances can be duplicated. In ot words, it's legal to duplicate a duplicate.

There are two ways to duplicate an instance using *duplicateMovieClip( )* :

- We can invoke *duplicateMovieClip ( )* as a *global function* , using the following syntax:
  `duplicateMovieClip(target, newName, depth);`

  where `target` is a string indicating the name of the instance we want to duplicate, `newName` is a string that specif the identifier for the new instance, and `depth` is an integer that designates where in the *content stack* (we'll discuss tl content stack soon) we want to place the new instance.

- We can also invoke *duplicateMovieClip ( )* as a method of an existing instance:
  `theClip.duplicateMovieClip(newName, depth);`

  where `theClip` is the name of the clip we wish to duplicate, and `newName` and `depth` both operate as in the previous example.

  When created using the *duplicateMovieClip( )* function, the newly created clip is attached as a child of either `target` 's or `theClip` 's parent. That is, the new clip becomes a sibling of either `target` or `myClip` in the movie clip hierarchy.

When created via *duplicateMovieClip( )* , an instance is initially positioned directly on top of its seed clip. Our first post-duplication task, therefore, is usually moving the duplicated clip to a new position. For example:

```
ball_mc.duplicateMovieClip("ball2_mc", 0);
ball2_mc._x += 100;
ball2_mc._y += 50;
```

Duplicated instances whose seed clips have been transformed (e.g., colored, rotated, or resized), via ActionScript or manually the Flash authoring tool, inherit the transformation of their seed clips at duplication time. Subsequent transformations to the

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

seed clip do not affect duplicated instances. Likewise, each instance can be transformed separately. For example, if a seed clip rotated 45 degrees and then duplicated, the duplicate instance's initial rotation is 45 degrees:

```
seedClip_mc._rotation = 45;
seedClip_mc.duplicateMovieClip("newClip_mc", 0);
trace(newClip_mc._rotation);                    // Displays: 45
```

If we then rotate the duplicate instance by 10 degrees, its rotation is 55 degrees, but the seed clip's rotation is still 45 degrees:

```
newClip_mc._rotation += 10;
trace(newClip_mc._rotation);      // Displays: 55
trace(seedClip_mc._rotation);     // Displays: 45
```

By duplicating many instances in a row and adjusting the transformation of each duplicate slightly, we can achieve interesting compound effects, such as stars in the sky, trails effects, and geometric animations (for an example, see "Load Event Starfield" the online Code Depot).

Using *duplicateMovieClip( )* offers other advantages over placing clips manually in a movie, such as the ability to:

- Control exactly when a clip appears on the Stage, relative to a program's execution
- Control exactly when a clip is removed from the Stage, relative to a program's execution
- Copy a clip's event handlers

These abilities give us advanced programmatic control over the content in a movie. With a manually created clip, we must preordain the birth and death of the clip using the timeline.

13.3.2.3 Creating instances with attachMovie( )

Like *duplicateMovieClip( )* , the *attachMovie( )* method lets us create a movie clip instance at runtime; however, *attachMovie* creates a new instance from a Library symbol instead of from another movie clip instance. To "attach" a movie clip means to make one movie clip the child of either the main timeline or another movie clip in the movie clip hierarchy. In order to use *attachMovie( )* to create an instance of a symbol, we must first *export* that symbol from the Library. Here's how:

1. In the Library, select the desired symbol.
2. In the Library's pop-up Options menu, select Linkage. The Linkage Properties dialog box appears.
3. Select the Export For ActionScript checkbox.
4. In the Identifier field, type a unique name for the clip symbol. The name can be any string—often simply the same nam the symbol itself—but should be different from all other exported clip symbols.
5. In Flash MX, we can also set the frame at which the link clip will be exported with the movie. For details, see *MovieClip.attachMovie ( )* in the ActionScript Language Reference.
6. Click OK.

Once a clip symbol has been exported, we can attach new instances of that symbol to an existing clip by invoking *attachMovi* with the following syntax:

```
theClip.attachMovie(symbolIdentifier, newName, depth, [initObject]);
```

where `theClip` is the name of the movie clip to which we want to attach the new instance. If `theClip` is omitted, *attachMovie( )* attaches the new instance to the current clip (the clip on which the *attachMovie( )* statement resides). The `symbolIdentifier` parameter is a string containing the name of the symbol we're using to generate our instance, as

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

specified in the Identifier field of the Linkage options in the Library (Step 4). The `symbolIdentifier` is not necessar the same name as the symbol itself. The `newName` parameter is a string that specifies the identifier for the new instance we creating. If `newName` is not a string that converts to a legal identifier, you're in for some potentially surprising results, as sh in the examples later in this section. Finally, `depth` is an integer that designates where in the host clip's content stack to pla the new instance. Flash MX adds support for a fourth, optional parameter called `initObject` , which lets you copy properties from an existing object to the new clip. See *MovieClip.attachMovie( )* in the ActionScript Language Reference for details.

For example, to attach a new movie clip, based on the "square" symbol, to the current timeline, use:

```
this.attachMovie("square", "square1_mc", 1);
square1_mc._x = 50;   // Position the new clip at a horizontal position of 50
square1_mc._y = 200;  // ...and a vertical position of 200
```

The *attachMovie( )* method attaches the new clip as a child of `theClip` (or of the current clip if no clip is specified). Contrast this with *duplicateMovieClip( )*, which attaches the clip as a sibling of the duplicated clip.

When we attach an instance to another clip, the instance is positioned in the center of the clip, among the clip's content stack. When we attach an instance to the main movie of a document, the instance is positioned in the upper-left corner of the Stage, coordinates (0, 0).

In Flash Player 6, the *attachMovie( )* function returns a reference to the newly created movie clip, which is useful for debugg If the function returns `undefined` , then the movie clip creation failed (except in Flash 5, where *attachMovie( )* always returns `undefined`). In this case, you most likely specified the wrong name for `symbolIdentifier` (remember to specify the quotes around the string, enter the name correctly, and use the export name specified in the Linkage options, not tl symbol's Library name). The `symbolIdentifier` string is not case-sensitive, so if the symbol is exported as "Symbol 1 you can (but should not, as a matter of good form) specify "symbol 1" as `symbolIdentifier` .

The return value of the *attachMovie( )* function is particularly useful if the operation succeeds and yet you are still having trou referring to your newly created movie clip by the expected identifier. Technically, `newName` does not need to be a legal identifier, but you'll have trouble referring to the new clip if it isn't, unless you store the return value in a variable. That is, you can use the return value to access the clip after it is attached, if the `newName` parameter was specified incorrectly and could be converted to a legal identifier. The `newName` parameter is actually converted first to a string, and later converted again t identifier. Let's see how this works.

In our first example, `newName` is specified as "newClip 1", which has a space in it, so it can't be converted to a legal identifier. Luckily, you can use the return value stored in `newClip_mc` to refer to the newly created clip:

```
newClip_mc = _root.attachMovie("Symbol 1", "newClip 1", 0);
trace(newClip_mc);  // Displays: _level0.newClip 1
                    // (which is not a valid identifier)
```

In the next example, `newName` is specified as `newClip1` , which takes the form of a legal identifier but is missing the necessary quotes to make it a string. The *attachMovie( )* command first tries to convert it to a string, but because `newClip1` looks to the interpreter like an `undefined` variable, the conversion yields the empty string. Luckily, you can again use the return value stored in `newClip_mc` to refer to the newly created clip:

```
newClip_mc = _root.attachMovie("Symbol 1", newClip1, 0);
trace(newClip_mc);  // Displays _level0.
                    // (the clip's name is an empty string!)
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

What if *newName* is specified as a legal identifier, such as `ball_mc`, that already refers to an existing movie clip? Again *attachMovie( )* command first tries to convert it to a string, which (referring to Table 3-2 for string conversion of movie clips) yields the interim string "_level0.ball_mc" for *newName*. This series of conversions results in the newly created movie cli having the incorrect name "_level0._level0.ball_mc". Luckily, yet again you can use the return value stored in `newClip_r` to refer to the newly created clip properly:

```
//Assume ball_mc is an existing (valid) movie clip identifier
newClip_mc = _root.attachMovie("Symbol 1", ball_mc, 0);
trace(newClip_mc);        // Displays "_level0._level0.ball_mc"
```

As you can see, life is much easier when we make sure to provide a string that converts to a legal identifier for *newName*.

13.3.2.4 Creating instances with createEmptyMovieClip( )

As of Flash Player 6, completely blank new movie clip instances can be created in an existing clip with the *createEmptyMovieClip( )* method, which has the following syntax:

```
theClip.createEmptyMovieClip(newName, depth);
```

where *theClip* is the name of an existing movie clip to which we want to attach a new, empty movie clip. The new clip instance is given a name of *newName* and placed in *theClip*'s content stack at the specified *depth*.

Movie clips created with *createEmptyMovieClip( )* are not derived from a Library symbol; they are simply blank movie clip instances with one frame. Empty movie clips can be used as drawing canvases, script clips, or containers for nested clips (e.g. form clip that contains text fields and a Submit button).

13.3.3 Movie Clip Instance Names

When we create instances, we assign them identifiers, or *instance names*, that allow us to refer to them later. Assigning identifiers to movie clips differs from assigning them to regular objects. When we create a typical object (not a movie clip), w must assign that object to a variable or other data container in order for the object to persist and in order for us to refer to it by name in the future. For example:

```
new Object( );                 // Object dies immediately after it's created, and
                               // we can't refer to it because we didn't store it.
var thing = new Object( );     // Object reference is stored in thing,
                               // and can later be referred to as thing.
```

Movie clip instances need not be stored in variables in order for us to refer to them. Unlike typical objects, movie clip instanc are accessible in ActionScript via their instance names as soon as they are created, either programmatically or in the authoring tool. The manner in which an instance gets its initial name depends on how it was created. Programmatically generated instan are named at runtime by the function that creates them. Manually created instances are normally assigned explicit instance na in the authoring tool through the Property inspector, as follows:

1. Select the instance on stage.
2. In the Property inspector, for <Instance Name>, enter the instance name.

(In Flash 5, the instance name is set via the Instance panel.) Once the instance is named in the authoring tool, it can be accesse via ActionScript using the same name. (It is good practice to add "_mc" as a suffix to the identifier name of any movie clips y create during authoring or at runtime.) For example, if there exists an instance named `ball_mc` on stage, we can access its properties like this:

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

```
ball_mc._y = 200;
```

If a manually created clip is not given an instance name, it is assigned one automatically by the Flash Player at runtime. Automatic instance names fall in the sequence `instance1` , `instance2` , `instance3` , ...`instance` *`n`* , but these names don't meaningfully describe our clip's content (and we must guess at the automatic name that was generated). For exan the first unnamed clip instance can be accessed as:

```
instance1._y = 200;
```

Because instance names assigned during either authoring or runtime are used as identifiers in ActionScript, we should always compose them according to the rules for creating a legal identifier, as described in Chapter 15. Most notably, instance names should not begin with a number or include hyphens or spaces. By convention, movie clip instance names should include the suffix "_mc", particularly if they are going to be referenced via ActionScript at runtime.

Each clip's instance name is stored in its built-in `_name` property, which can be both retrieved and set. For clips defined manually during authoring, the default `_name` is a string version of the original clip identifier:

```
trace(ball_mc._name);       // Displays "ball_mc"
```

For programmatically defined clips, the initial value for `_name` is specified by the *`newName`* parameter passed to *duplicateMovieClip( )* or *attachMovie( )* .

The `_name` property is useful for debugging or displaying the name of a clip (without the fully qualified path). Note how the value returned by `_name` differs from other representations of a movie clip identifier (see also the legacy `_target` propert discussed later in this chapter):

```
trace(instance1._name);          // Displays instance1
trace(instance1);                // Displays _level0.instance1
trace(String(instance1));        // Displays _level0.instance1
trace(targetPath(instance1));    // Displays _level0.instance1
trace(instance1.toString( ));     // Displays [object Object]
```

Example 13-1 uses the `_name` property to find a particular movie clip. See Section 13.5.6.4 later in this chapter for an example showing how to use the `_name` property to prevent an infinite loop.

Example 13-1. Finding movie clips on a timeline

```
// Finds all movie clips inside gameboard_mc with the word "enemy" in their nai
for (var prop in gameboard_mc) {
  if (typeof gameboard_mc[prop] =  = "movieclip") {
    if (gameboard_mc[prop]._name.indexOf("enemy") != -1) {
      // Found an enemy movie clip...make it attack the player.
      gameboard_mc[prop].attackPlayer( );
    }
  }
}
```

If we change an instance's `_name` property, all future references to the instance must use the new name. For example, if we change the value of `ball_mc._name` , the `ball_mc` reference ceases to exist, and we must subsequently use the new na to refer to the instance:

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

```
ball_mc._name = "circle_mc";  // Change ball_mc's name to circle_mc
trace(typeof ball_mc);        // Displays "undefined" because ball_mc
                              // no longer exists.
circle_mc._x = 59;            // After the name change, you must
                              // use the clip's new name.
```

Therefore, you shouldn't change a movie clip's `_name` property at runtime, as it is can make your code fail, or at least make i very difficult to follow.

13.3.4 Importing External Movies and Images

We've discussed creating movie clip instances within a single document, but the Flash Player can also display multiple *.swf* documents simultaneously. We can use *loadMovie( )* — as either a global function or a movie clip method — to import an external *.swf* file into the Player and place it either in a clip instance or on a numbered level above the base movie (i.e., in the foreground relative to the base movie).

In Flash Player 6, *loadMovie( )* can also load JPEG image files into a movie clip or document level. For details, see *loadMovie( )* and *MovieClip.loadMovie( )* in the *Language Reference* .

Dividing content into separate files gives us precise control over the downloading process and makes partial application upda easier. Suppose, for example, we have a movie containing a main navigation menu and five subsections. Before the user can navigate to section five, sections one through four must finish downloading. But if we place each section in a separate *.swf* fi the sections can be loaded in an arbitrary order, giving the user direct access to each section. To update a section, we can simp replace the appropriate *.swf* file with a new one.

When an external *.swf* is loaded into a level, its main movie timeline becomes the root timeline of that level, and it replaces a prior movie loaded in that level. Similarly, when an external movie is loaded into a clip, the main timeline of the loaded movi replaces that clip's timeline, unloading the existing graphics, sounds, and scripts in that clip.

Like *duplicateMovieClip( )* , *loadMovie( )* can be used as both a standalone function and an instance method. The standalone syntax of *loadMovie( )* is as follows:

```
loadMovie(url, location)
```

where `url` specifies the address of the external *.swf* file to load. The `location` parameter is a string indicating the path an existing clip or a document level that should host the new *.swf* file (i.e., where the loaded movie should be placed). For example:

```
loadMovie("circle.swf", "_level1");
loadMovie("photos.swf", "viewClip_mc");
```

Because a movie clip reference is converted to a path when used as a string, `location` can also be supplied as a movie cli reference, such as `_level1` instead of `"_level1"` . Take care when using references, however. If the reference supplied does not point to a valid clip, the *loadMovie( )* function has an unexpected behavior—it loads the external *.swf* into the *currer* timeline. See Section 13.7 later in this chapter for more information on this topic.

The *MovieClip* method version of *loadMovie( )* has the following syntax:

```
theClip.loadMovie(url);
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

When used as a clip method, *loadMovie( )* assumes we're loading the external *.swf* into `theClip` , so the `location` parameter required by the standalone *loadMovie( )* function is not needed. Therefore, we supply only the path to the *.swf* to lo via the `url` parameter. Naturally, `url` can be either an absolute or a relative filename, such as:

```
viewClip.loadMovie("photos.swf");
```

When placed into a clip instance, a loaded movie adopts the properties of that clip (e.g., the clip's scale, rotation, color transformation, etc.).

Note that `theClip` must exist in order for *loadMovie( )* to be used in its method form. For example, the following attempt load *circle.swf* will fail if `_level1` is empty:

```
_level1.loadMovie("circle.swf");
```

Using loadMovie( ) with attachMovie( )

Loading an external *.swf* file into a clip instance with *loadMovie( )* has a surprising result—it prevents us from attaching instances to that clip via *attachMovie( )* . Once a clip has an external *.swf* file loaded into it, that clip can no longer bear attached movies from the Library from which it originated. For example, if *movie1.swf* contains an instance named `clipA` , and we load *movie2.swf* into `clipA` , we can no longer attach instances to `clipA` from *movie1.swf* 's Library.

Why? The *attachMovie( )* method works only within a *single* document. That is, we can't attach instances from one document's Library to another document. When we load a *.swf* file into a clip, we are populating that clip with a new document and, hence, a new (different) Library. Subsequent attempts to attach instances from our original document to the clip fail, because the clip's Library no longer matches its original document's Library. However, if we unload the document in the clip via *unloadMovie( )* , we regain the ability to attach movies to the clip from its own document Library.

Similarly, loading a *.swf* file into a clip with *loadMovie( )* prevents us from copying that clip via *duplicateMovieClip( )* .

13.3.4.1 Load movie execution order

The *loadMovie( )* function is not immediately executed when it appears in a statement block. In fact, it is not executed until al other statements in the block have finished executing.

> We cannot access an externally loaded movie's properties or methods in the same statement block as the *loadMovie( )* invocation that loads it into the Player.

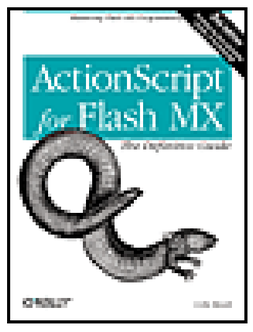
Because *loadMovie( )* loads an external file (usually over a network), its execution is *asynchronous* . That is, *loadMovie( )* ma finish at any time, depending on the speed of the file transfer. Therefore, before we access a loaded movie, we should always check that the movie has finished transferring to the Player. We do so with what's commonly called a *preloader* —code that checks how much of a file has loaded before allowing some action to take place. Preloaders can be built with the `_totalframes` and `_framesloaded` movie clip properties and the *getBytesLoaded( )* and *getBytesTotal( )* movie clip methods. See the appropriate entries under the *MovieClip* class in the *Language Reference* for sample code.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir



ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Chapter 13. Movie Clips

13.4 Movie and Instance Stacking Order

All movie clip instances and externally loaded movies displayed in the Player reside in a visual stacking order akin to a deck of cards. When instances or externally loaded *.swf* files overlap in the Player, one clip (the "higher" of the two) obscures the other clip (the "lower" of the two). This appears simple enough in principle, but the main *content stack*, which contains all the instances and *.swf* files, is actually divided into many smaller substacks. We'll first look at these substacks individually first, and then we'll see how they combine to form the main stack. (The content stack in this discussion has no direct relation to the LIFO and FIFO stacks discussed in Chapter 11.)

13.4.1 Movie Clip Depths

Each movie clip instance, including the main timeline of a movie, places its various contents (movie clips, text fields, and buttons) on one of two stacks: the internal layer stack (for author-time assets) or the programmatically generated content stack (for runtime assets). The items in these stacks (known collectively as the *clip's content stack*) are given an integer *depth* position that governs how they overlap on screen. Depth positions range from -16384 to 1048575. Depths from 0 to 1048575 are reserved for dynamically generated content; depths from -16383 to -1 are reserved for author-time content; and depth -16384 is reserved for dynamic content that appears beneath all author-time content in each clip. To retrieve the depth position of an item, we use the *getDepth( )* method. To change the depth position of two movie clips, we use the *swapDepths( )* method.

13.4.2 The Internal Layer Stack

Instances created manually in the Flash authoring tool reside in the *internal layer stack*. This stack's order is governed by the actual layers in a movie's timeline; when two manually created instances on separate timeline layers overlap, the instance on the uppermost layer obscures the instance on the lowermost layer. (Here, "uppermost" means that a layer appears at the top of the timeline panel in the Flash authoring tool).

Furthermore, because multiple clips can reside on a single timeline layer, each layer in the internal layer stack actually maintains its own ministack. Overlapping clips that reside on the same layer of a timeline are stacked in the authoring tool via the Modify Arrange commands.

We can swap the position of two instances in the internal layer stack using the *swapDepths( )* method, provided they reside on the same timeline (that is, the value of the two clips' `_parent` property must be the same). Prior to Flash 5, there was no way to alter the internal layer stack via ActionScript.

The depth position of author-time assets can be unpredictable and is considered reserved for use by the authoring tool. Therefore, when swapping the depths of an author-time movie clip and a dynamically created clip, always use the *target* parameter of *swapDepths( )* — not an integer depth level — as described in the ActionScript Language Reference entry for *MovieClip.swapDepths( ).*

13.4.3 The Programmatically Generated Content Stack

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Programmatically generated instances are normally stacked separately from the *manually* created instances held in the internal layer stack. Each movie clip has its own *programmatically generated content stack* that holds:

- Movie clip instances created via *duplicateMovieClip( )*, *attachMovie( )*, and *createEmptyMovieClip( )*
- Text field instances created via *createTextField( )*

The stacking order for movie clips in the programmatically generated content stack varies, depending on how they were created.

13.4.3.1 How clips generated via attachMovie( ) and createEmptyMovieClip( ) are added to the stack

A new instance generated via *attachMovie( )* or *createEmptyMovieClip( )* is always stacked above (i.e., in the foreground relative to) the clip to which it was attached. For example, suppose that we have two clips—`X` and `Y`—in the internal layer stack of a movie and that `X` resides on a layer above `Y`. Now further suppose we attach a new clip, `A`, to `X` and a new clip, `B`, to `Y`:

```
x.attachMovie("A", "A", 0);
y.attachMovie("B", "B", 0);
```

In our scenario, the clips appear from top to bottom in this order: `A`, `X`, `B`, `Y`, as shown in Figure 13-1.

Figure 13-1. A sample instance stack

Once a clip is generated, it too provides a separate space above its internal layer stack for more programmatically generated clips. That is, we can attach clips to attached clips.

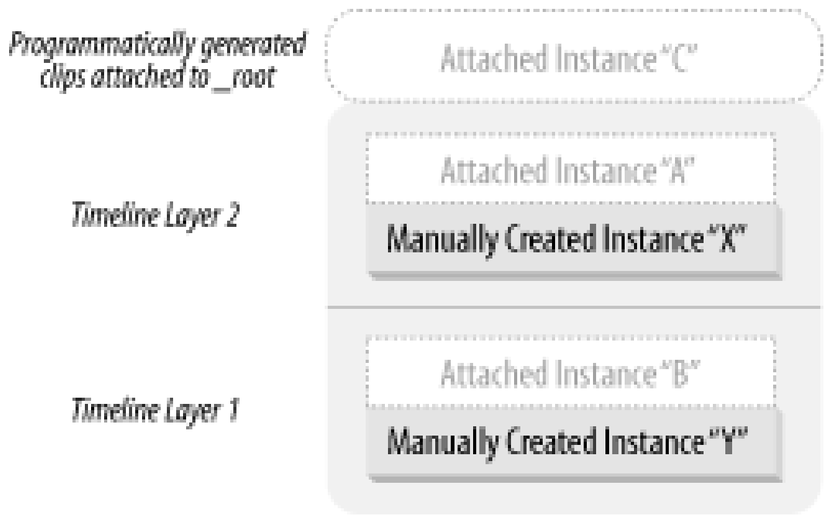
Clips attached to the `_root` movie of a Flash document are placed in the `_root` movie's programmatically generated content stack, which appears in front of *all* clips in the `_root` movie, even those that contain programmatically generated content.

Let's extend our example. If we attach clip `C` to the `_root` of the movie that contains clips `X`, `Y`, `A`, and `B`, then clip `C` appears in front of all the other clips. Figure 13-2 shows the extended structure.

Figure 13-2. An instance stack showing a clip attached to _root

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir



13.4.3.2 How clips generated via duplicateMovieClip( ) are added to the stack

Each instance duplicated via *duplicateMovieClip( )* is assigned to the programmatic stack of its seed clip's parent (the movie clip upon whose timeline the seed clip resides). Let's return to our example to see how this works.

If we create clip `D` by duplicating clip `X` (which was created manually), then clip `D` is placed in the stack above `_root`, with clip `C`. A seed clip and its duplicate always share the same parent (in this example, `_root`). Similarly, if we create clip `E` by duplicating clip `D`, then `E` is also placed in the stack above `_root`, with `C` and `D`. But if we create clip `F` by duplicating clip `A` — which was created with *attachMovie( )* — then `F` is placed in the programmatic stack above `X`, with clip `A`. Again, `F` and its seed clip, `A`, share the same parent: `X`. Figure 13-3 is worth a thousand words.

Figure 13-3. An instance stack showing various duplicated clips

13.4.3.3 Assigning depths to instances in the programmatically generated content stack

You may be wondering what determines the stacking order of clips `C`, `D`, and `E`, or of clips `A` and `F`, in Figure 13-3. The stacking order of a programmatically generated clip is determined by the `depth` argument passed to the *attachMovie( )*, *createEmptyMovieClip( )*, or *duplicateMovieClip( )* methods, and can be changed at any time using the *swapDepths( )* function. Each programmatically generated clip's `depth` (sometimes called its *z-index*) determines its position within a particular stack of programmatically generated clips.

The `depth` of a clip can be any integer and is measured from the bottom up — so, depth 5 is lower than depth 6,

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

depth 7 is higher than (i.e., in front of) depth 6, depth 8 is higher still, and so on. When two programmatically generated clips occupy the same position on screen, the one with the greater `depth` value is rendered in front of the other.

Although multiple clips can occupy a single author-time timeline layer, all depth positions are single-occupant dwellings. Only one clip can occupy a depth position at a time—placing a clip into an occupied position displaces (and deletes) the layer's previous occupant.

It's okay for there to be gaps in the depths of clips; you can have a clip at depth 0, another at depth 500, and a third one at depth 1000. No performance hit or increase in memory consumption results from having gaps in your depth assignments.

13.4.4 The .swf Document "_level" Stack

In addition to the internal layer stack and the programmatically generated content stack, there's a third (and final) kind of stack, the *document stack* (or *level stack*), which governs the overlapping not of instances, but of entire *.swf* files loaded into the Player via *loadMovie( )*.

The first *.swf* file loaded into the Flash Player is placed in the lowest level of the document stack (represented by the global property `_level0`). If we load any additional *.swf* files into the Player after that first document, we can optionally place them in front of the original document by assigning them to a level above `_level0` in the document stack. All of the content in the higher-level documents in the level stack appears in front of lower-level documents, regardless of the movie clip stacking order within each document.

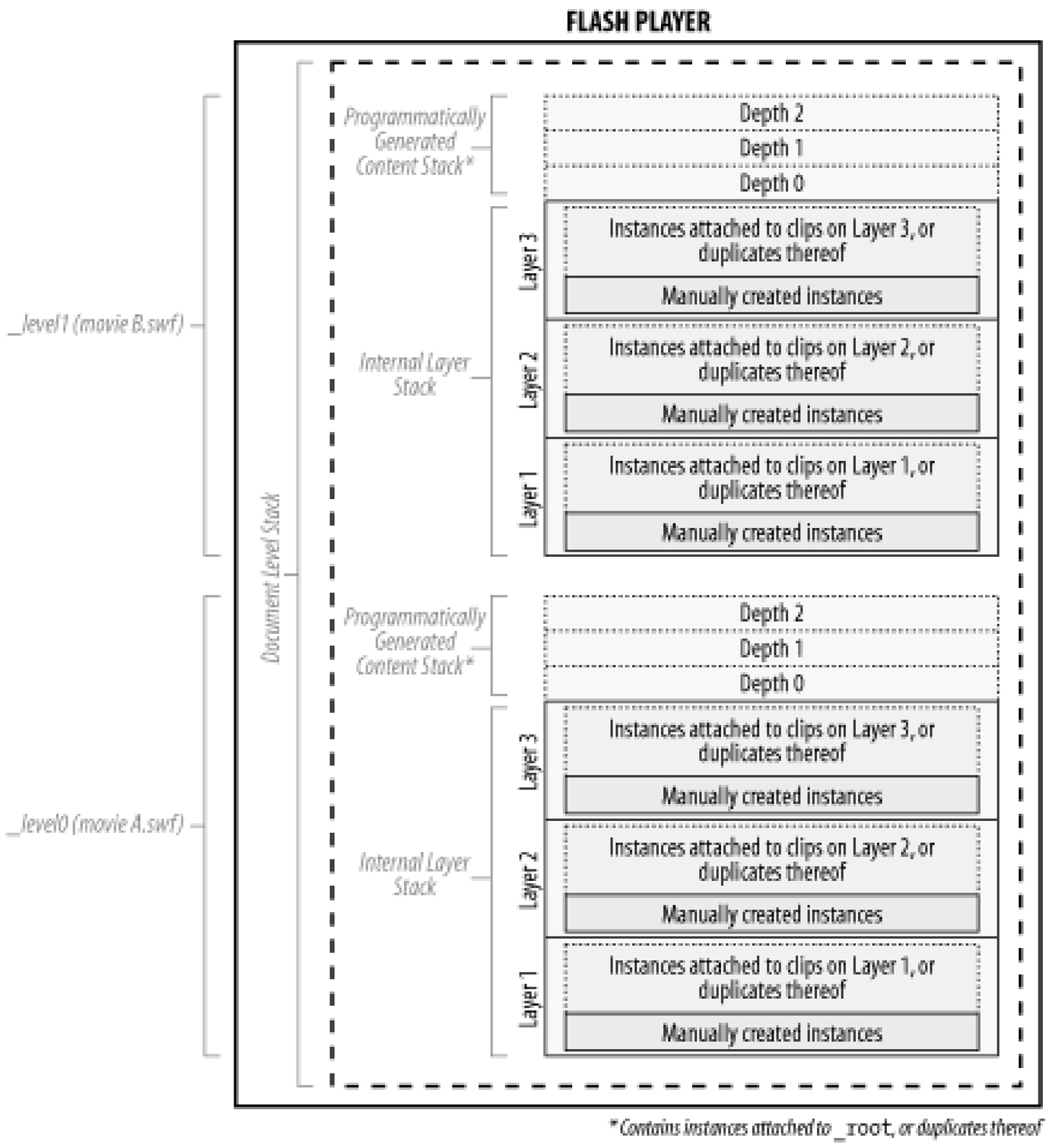
Just as the programmatically generated content stack allows only one clip per layer, the document stack allows only one document per level. If we load a *.swf* file into an occupied level, the level's previous occupant is replaced by the newly loaded document. For example, you can supplant the original document by loading a new *.swf* file into `_level0`. Loading a new *.swf* file into `_level1` visually obscures the movie in `_level0`, but it does not remove it from the Player.

Figure 13-4 summarizes the relationships of the various stacks maintained by the Flash Player.

Figure 13-4. The complete Flash Player content stack

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir



13.4.5 Stacks and Order of Execution

The layering of movie clips and timeline layers affects code execution order. The rules are as follows:

- Code on frames in different timeline layers always executes from top to bottom (relative to the timeline panel).
- When manually created instances are initially loaded, code in their timeline and *onLoad( )* event handlers executes according to the Load Order set in the Publish Settings of a Flash document—either Bottom Up, which is the default, or Top Down.

  For example, suppose we have a timeline with two layers, *top* and *bottom*, where *top* is above *bottom* in the layer stack. We place clip X on layer *top* and clip Y on layer *bottom*. If the Load Order of the document is set to Bottom Up, then the code in clip Y will execute before the code in clip X. If, on the other hand, the Load Order of the document is set to Top Down, then the code in clip X will execute before the code in clip Y. This execution order applies *only* to the frame on which X and Y appear for the first time.

- Once loaded, all instances of a movie are added to an execution order, which is the reverse of the load order; the last instance added to the movie is always the first to have its code executed.

Use caution when relying on these rules. Layers are mutable, so you should avoid producing code that relies on their

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

relative position. Strive to create code that executes safely without relying on the execution order of the clips in the stack. We can avoid some of the issues presented by the execution stack by keeping all our code on a *scripts* layer at the top of each code-bearing timeline.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Chapter 13. Movie Clips

13.5 Referring to Instances and Main Movies

In earlier sections of this chapter, we saw how to create and layer movie clip instances and external *.swf* files in the Flash Pla
We must be able to refer to that content in order to effectively control it with ActionScript.

We refer to instances and main movies under four general circumstances, when we want to:

- Get or set a property of a clip or a movie
- Create or invoke a method of a clip or a movie
- Apply some function to a clip or a movie
- Manipulate a clip or movie as data — for example, by storing it in a variable or passing it as an argument to a function

While the circumstances under which we refer to clip instances and movies are fairly simple, the tools we have for making references are many and varied. We'll spend the rest of this section exploring ActionScript's instance- and movie-referencing

13.5.1 Using Instance Names

Earlier, we saw that movie clips are referred to by their *instance names* . For example:

```
trace(someVariable);     // Refer to a variable
trace(someClip_mc);      // Refer to a movie clip
```

In order to refer to an instance directly (as shown in the preceding *trace( )* example), the instance must reside on the timeline t
which our code is attached. For example, if we have an instance named `clouds_mc` placed on the main timeline of a docur
we can refer to `clouds_mc` from code attached to the main timeline, as follows:

```
// Set a property of the instance
clouds_mc._alpha = 60;
// Invoke a method on the instance
clouds_mc.play();
// Place the instance in an array of other related instances
var background = [clouds_mc, sky_mc, mountains_mc];
```

If the instance we want to reference does not reside on the same timeline as our code, we must use a more elaborate syntax, as
described later in this section under Section 13.5.3.

13.5.2 Referring to the Current Instance or Movie

We don't always have to use an instance's name when referring to a clip. Code attached to a frame in an instance's timeline ca
refer to that instance's properties and methods directly, without any instance name.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

For example, to set the `_alpha` property of a clip named `cloud_mc` , we can place the following code on a frame in the `cloud_mc` timeline:

```
_alpha = 60;
```

Similarly, to invoke the *play( )* method on `cloud_mc` from a frame in the `cloud_mc` timeline, we can simply use:

```
play( );
```

This technique can be used on any timeline, including timelines of main movies. For example, the following two statements a synonymous if attached to a frame on the main timeline of a Flash document. The first refers to the main movie implicitly, wl the second refers to the main movie explicitly via the global `_root` property:

```
gotoAndStop(20);
_root.gotoAndStop(20);
```

However, not all methods can be used with an implicit reference to a movie clip. Any movie clip method that has the same na a corresponding global function, such as *duplicateMovieClip( )* or *unloadMovie( )* , must be invoked with an explicit instance reference. Hence, when in doubt, use an explicit reference. We'll have more to say about method and global function conflicts later in in this chapter under Section 13.7.

Note that it's always safest to use explicit references to variables or movie clips rather than using implicit references. Implicit references are ambiguous, often causing unexpected results and confusing other developers reading your code.

13.5.2.1 Self-references with the this keyword

When we want to refer explicitly to the current instance from a frame in its timeline or from one of its event handlers, we can the `this` keyword. For example, the following statements are synonymous when attached to a frame in the timeline of our `cloud_mc` instance:

```
_alpha = 60;        // Implicit reference to the current timeline
this._alpha = 60;   // Explicit reference to the current timeline
```

There are three reasons to use `this` to refer to a clip even when we could legitimately refer to the clip's properties and meth directly.

- First, explicit references are easier for other developers to read, because they make the intention of a statement unambig
- Second, when used without an explicit instance reference, certain movie clip methods are mistaken for global functions the interpreter. If we omit the `this` reference, the interpreter thinks we're trying to invoke the analogous global functi and complains that we're missing the *target* movie clip parameter. To work around the problem, we use `this` , as follc

  ```
  this.duplicateMovieClip("newClouds_mc", 0);  // Invoke a method on an inst

  // If we omit the this reference, we get an error
  duplicateMovieClip("newClouds_mc", 0);       // Oops!
  ```

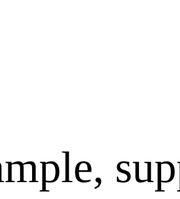
- Third, using `this` , we can conveniently pass a reference to the current timeline to functions that operate on movie cli

  ```
  // Here's a function that manipulates clips
  function moveClipTo (theClip, x, y) {
    theClip._x = x;
    theClip._y = y;
  ```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

```
}

// Now let's invoke it on the current timeline
moveClipTo(this, 150, 125);
```



New and experienced object-oriented programmers alike, take note: the meaning of `this` inside a method is a reference not to the current timeline but to the object through which the method was invoked. If an object's method needs to refer to a specific movie clip, a reference to that clip should be passed to the method as a parameter. See Chapter 12 for details on using the `this` keyword inside methods.

13.5.3 Referring to Nested Instances

As we discussed in the introduction to this chapter, movie clip instances are often nested inside of one another. That is, a clip' canvas can contain an instance of another clip, which can itself contain instances of other clips. For example, a game's `spaceship` clip can contain an instance of a `blinkingLights` clip or a `burningFuel` clip. Or a character's `face` can include separate `eyes` , `nose` , and `mouth` clips.

Earlier, we saw briefly how we can navigate up or down from any point in the hierarchy of clip instances, much like you navi up and down a series of subdirectories on your hard drive. Let's examine this in more detail and see some more examples.

Let's first consider how to refer to a clip instance that is nested inside of the current instance.

> When a clip is placed on the timeline of another clip, it becomes a property of that clip, and we can access it as we would access any object property (with the dot operator).

For example, suppose we place `clipB` on the canvas of `clipA` . To access `clipB` from a frame in `clipA` 's timeline, we direct reference to `clipB` :

```
clipB._x = 30;
```

We could also use an explicit reference, as in:

```
this.clipB._x = 30;
```

Now suppose `clipB` contains another instance, `clipC` . To refer to `clipC` from a frame in `clipA` 's timeline, we access `clipC` as a property of `clipB` , like this:

```
clipB.clipC.play();
clipB.clipC._x = 20;
```

Beautiful, ain't it? And the system is infinitely extensible. Because every clip instance placed on another clip's timeline becon property of its host clip, we can traverse the hierarchy by separating the instances with the dot operator, like so:

```
clipA.clipB.clipC.clipD.gotoAndStop(5);
```

Now that we've seen how to navigate down the instance hierarchy, let's see how we navigate up the hierarchy to refer to the instance or movie that contains the current instance. As we saw earlier, every instance has a built-in `_parent` property that to the clip or main movie containing it. We use the `_parent` property like so:

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

```
theClip._ parent
```

where *theClip* is a reference to a movie clip instance. Recalling our recent example with clipA on the main timeline, clipB inside clipA , and clipC inside clipB , let's see how to use _parent and dot notation to refer to the various c in the hierarchy. Assume that the following code is placed on a frame of the timeline of clipB :

```
_parent           // A reference to clipA
this              // An explicit relative reference to clipB (the current clip)
this._parent      // An explicit relative reference to clipA
// Sweet Sheila, I love this stuff! Let's try some more...
_parent._parent   // A reference to clipA's parent (clipB's grandparent),
                  // which is the main timeline in this case
```

Note that although it is legal to do so, it is unnecessarily roundabout to traverse *down* the hierarchy using a reference to the clipC property of clipB only to traverse back *up* the hierarchy using _parent . These roundabout references are unnecessary but do show the flexibility of dot notation:

```
clipC._parent                     // A roundabout reference to clipB
                                  // (the current timeline)
clipC._parent._parent._parent     // A roundabout reference to the main timeline
```

Notice how we use the dot operator to descend the clip hierarchy and use the _parent property to ascend it.

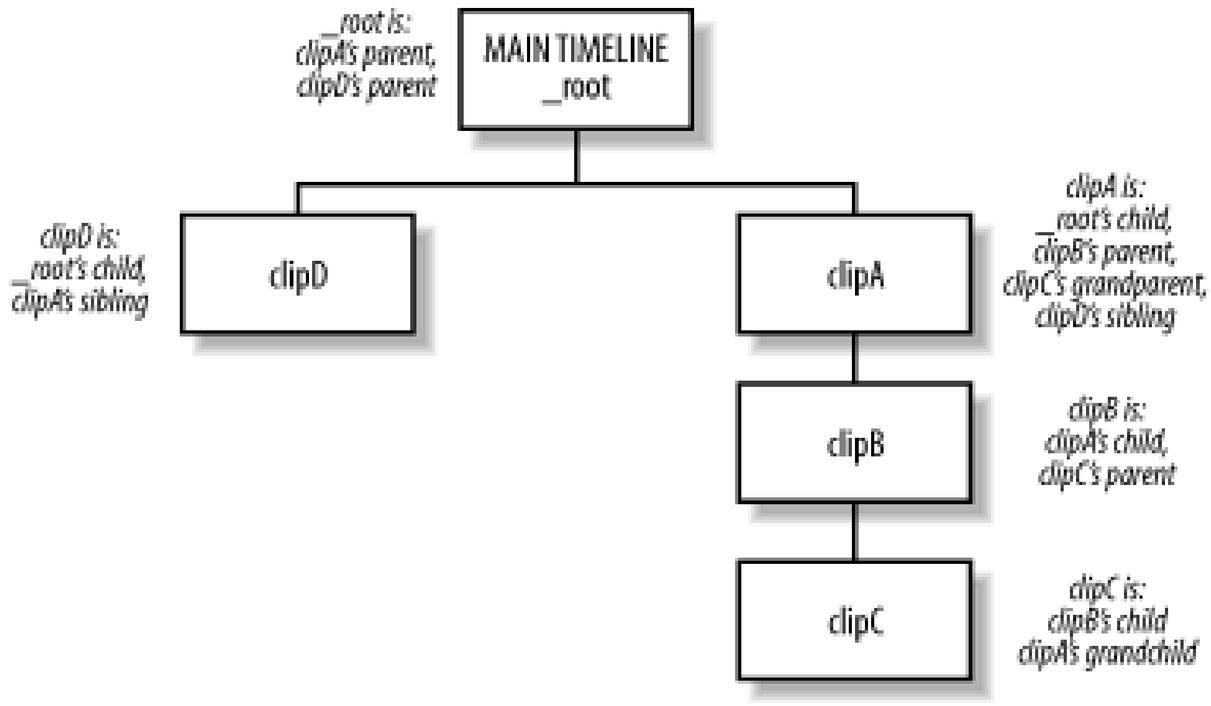
If this is new to you, you should probably build the clipA , clipB , clipC hierarchy in Flash and test the code in our example. Proper instance referencing is one of the fundamental skills of a good ActionScript programmer.

Note that the hierarchy of clips is like a family tree. Unlike a typical family tree of a sexually reproducing species, in which e offspring has two parents, our clip family tree expands asexually. That is, each household is headed by a single parent who ca adopt any number of children. Any clip (i.e., any *node* in the tree) can have one and only one parent (the clip that contains it) can have multiple *children* (the clips that it contains). Of course, each clip's parent can in turn have a single parent, which mea that each clip can have only one grandparent (not the four grandparents humans typically have). Figure 13-5 shows a sample c hierarchy.

Figure 13-5. A sample clip hierarchy

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir



No matter how far you go down the family tree, if you go back up the same number of steps you will always end up in the sar place you started. It is therefore pointless to go down the hierarchy only to come back up. However, it is *not* pointless to go u hierarchy and then follow a *different* path back down. For example, suppose that our example main timeline also contains `cl` which makes `clipD` a "sibling" of `clipA` because both have the main timeline as their `_parent` . In this case, you can re `clipD` from a script attached to `clipB` as follows:

```
_parent._parent.clipD    // This refers to clipD, a child of the main
                         // timeline (clipA's _parent) and therefore
                         // a sibling of clipA
```

Note that the main timeline does not have a `_parent` property (main movies are the top of any clip hierarchy and cannot be contained by another timeline); references to `_root._parent` yield `undefined` .

13.5.4 Referring to Main Movies with _root and _levelN

Now that we've seen how to navigate up and down the clip hierarchy *relative* to the current clip, let's explore other ways to navigate along *absolute* pathways and even among other documents stored in other levels of the Player's document stack. In e chapters, we saw how these techniques applied to variables and functions; here we'll see how they can be used to control mov clips.

13.5.4.1 Referencing the current level's main movie using _root

When an instance is deeply nested in a clip hierarchy, we can repeatedly use the `_parent` property to ascend the hierarchy we reach the main movie timeline. But in order to ease the labor of referring to the main timeline from deeply nested clips, w also use the built-in global property `_root` , which is a shortcut reference to the main movie timeline. For example, here we the main movie:

```
_root.play();
```

The `_root` property is said to be an *absolute reference* to a known point in the clip hierarchy because unlike the `_parent` `this` properties, which are relative to the current clip, the `_root` property refers to the main timeline of the current level, n matter which clip within the hierarchy references it (see the exception in the next warning). These are all equivalent (except fi scripts attached to the main timeline, where `_parent` is not valid):

```
_parent._root
this._root
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

`_root`

Therefore, you can use `_root` when you don't know where a given clip is nested within the hierarchy. For example, conside following hierarchy in which `circle` is a child of the main movie timeline and `square` is a child of `circle` :

```
main timeline
   circle
     square
```

Now consider this script attached to a frame in both `circle` and `square` :

```
_parent._x += 10  // Move this clip's parent clip 10 pixels to the right
```

When this code is executed from within `circle` , it causes the main movie to move 10 pixels to the right. When it is execut from within `square` , it causes `circle` (not the main movie) to move 10 pixels to the right. In order for the script to move main movie 10 pixels regardless of where the script is executed from, the script can be rewritten as:

```
_root._x += 10   // Move the main movie 10 pixels to the right
```

Furthermore, the `_parent` property is not valid from within the main timeline; the version of the script using `_root` is val even when used in a frame of the main timeline.

The `_root` property can be combined with ordinary instance references to descend a nested-clip hierarchy:

```
_root.clipA.clipB.play( );
```

References that start with `_root` refer to the same, known, starting point from anywhere in a document. There's no guessing required.

> When a *.swf* file is loaded into a movie clip instance, `_root` refers no longer to that *.swf* file's main timeline but to the main timeline of the movie into which the *.swf* was loaded!

If you know your movie will be loaded into a movie clip, you should not use `_root` to refer to the main timeline. Instead, de global reference to the main timeline by placing the following code on your movie's main timeline:

```
_global.myAppMain = this;
```

where *myApp* is the name of your application and `Main` is used, by convention, to denote the main timeline. Then use *myA* `Main` in place of `_root` .

13.5.4.2 Referencing other documents in the Player using _levelN

If we have loaded multiple *.swf* files into the document stack of the Flash Player using *loadMovie( )* , we can refer to the mai movie timelines of the various documents using the built-in series of global properties `_level0` through `_level`*n* , whe represents the level of the document we want to reference.

Therefore, `_level0` represents the document in the lowest level of the document stack (documents in higher levels will be rendered in the foreground). Unless a movie has been loaded into `_level0` via *loadMovie( )* , `_level0` is occupied by th movie that was initially loaded when the Player started.

Here is an example that plays the main movie timeline of the document in level 3 of the Player's document stack:

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

```
_level3.play();
```

Like the `_root` property, the `_level`*n* property can be combined with ordinary instance references via the dot operator:

```
_level1.clipA.stop();
```

As with references to `_root` , references to `_level`*n* properties are called *absolute references* because they lead to the s destination from any point in a document.

Note that `_level`*n* and `_root` are not synonymous. The `_root` property is always the *current* document's main timelin regardless of the level on which the current document resides, whereas the `_level`*n* property is a reference to the main timeline of a specific document level. For example, suppose we place the code *_root.play( )* in *myMovie.swf* . When we load *myMovie.swf* onto level 5, our code plays `_level5` 's main movie timeline. In contrast, if we place the code *_level2.play( )* in *myMovie.swf* and load *myMovie.swf* into level 5, our code plays `_level2` 's main movie timeline, not `_level5` 's. Of c from within level 2, `_root` and `_level2` are equivalent.

13.5.5 Authoring Instance References with Insert Target Path

When the instance structure of a movie gets very complicated, composing references to movie clips and main movies can be laborious. We may not always recall the exact hierarchy of a series of clips and, hence, may end up frequently selecting and editing clips in the authoring tool just to determine their nested structure. The Actions panel's Insert Target Path tool generate references visually, relieving the burden of creating them manually. The Insert Target Path button is shown in Figure 13-6.

Figure 13-6. The Insert Target Path button

To use Insert Target Path, follow these steps:

1. Position the cursor in your code where you want a clip reference to be inserted.
2. Click the Insert Target Path button, shown in Figure 13-6.
3. In the Insert Target Path dialog box, select the clip to which you want to refer.
4. Choose whether to insert an *absolute reference* , which begins with `_root` , or a *relative reference* , which expresses t reference to the target clip in relation to the clip that contains your code (`this`).
5. If you are exporting to Flash 4 format, choose the Slashes Notation button for Flash 4 compatibility. (The Dot Notation button, selected by default, composes references that won't work in Flash 4).

The Insert Target Path tool cannot generate relative references that ascend a hierarchy of clips. That is, the tool cannot be use refer to a clip that contains the current clip (unless you want to begin the path from `_root` and proceed downward). To crea references that ascend the clip hierarchy, we must either use absolute references starting with `_root` (which therefore becon

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

descending references) or manually enter the appropriate relative references in our code using the `_parent` property.

13.5.6 Dynamic References to Clip Objects

Normally, we know the name of the specific instance or movie we are manipulating, but there are times when we'd like to cor clip whose name we don't know. We may, for example, want to scale down a whole group of clips using a loop or create a bu that refers to a different clip each time it is clicked. To handle these situations, we must create our clip references dynamically runtime.

13.5.6.1 Using the array-element access operator

As we saw in Chapter 5 and Chapter 12, the properties of an object can be retrieved via the dot operator or through the array-element access operator, `[ ]` . For example, the following two statements are equivalent:

```
someObject.myProperty = 10;
someObject["myProperty"] = 10;
```

The array-element access operator has one important feature that the dot operator does not; it lets us (indeed requires us to) re a property using a *string expression* rather than an *identifier* . For example, here's a string concatenation expression that acts a valid reference to the property `propertyX` :

```
someObject["prop" + "ertyX"];
```

We can apply the same technique to create our instance and movie references dynamically. We already saw that clip instance stored as properties of their parent clips. Earlier, we used the dot operator to refer to those instance properties. For example, fi the main timeline we can refer to `clipB` — which is nested inside of another instance, `clipA` — as follows:

```
clipA.clipB;                    // Refer to clipB inside clipA
clipA.clipB.stop();             // Invoke a method on clipB
```

Because instances are properties, we can also legitimately refer to them with the `[ ]` operator, as in:

```
clipA["clipB"];                 // Refer to clipB inside clipA
clipA["clipB"].stop();          // Invoke a method on clipB
```

Notice that when we use the `[ ]` operator to refer to `clipB` , we provide the name of `clipB` as a string, not an identifier. string reference can be any valid string-yielding expression. For example, here's a reference to `clipB` that involves a string concatenation:

```
var clipCount = "B";
clipA["clip" + clipCount];          // Refer to clipB inside clipA
clipA["clip" + clipCount].stop();   // Invoke a method on clipB
```

We can create clip references dynamically to refer to a series of sequentially named clips:

```
// Here's a loop that stops clip1, clip2, clip3, and clip4
for (var i = 1; i <= 4; i++) {
  _root["clip" + i].stop( );
}
```

Now that's powerful!

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

13.5.6.2 Storing references to clips in data containers

We began this chapter by saying that though movie clips are technically their own datatype, they are treated as objects in ActionScript. Hence, we can store a reference to a movie clip instance in a variable, an array element, or an object property.

Recall our earlier example of a nested instance hierarchy (`clipC` nested inside `clipB` nested inside `clipA`) placed on th main timeline of a document. If we store these various clips in data containers, we can control them dynamically using the containers instead of explicit references to the clips. Example 13-2, which shows code that is placed on a frame in the main timeline, uses data containers to store and control instances.

Example 13-2. Storing clip references in variables and arrays

```
var x = clipA.clipB;   // Store a reference to clipB in the variable x
x.play();              // Play clipB
// Now let's store our clips in the elements of an array
var theClips = [clipA, clipA.clipB, clipA.clipB.clipC];
theClips[0].play();    // Play clipA
theClips[1]._x = 200;  // Place clipB 200 pixels from clipA's registration poi
// Stop all the clips in our array using a loop
for (var i = 0; i < myClips.length; i++) {
  myClips[i].stop();
}
```

By storing clip references in data containers, we can manipulate the clips (such as playing, rotating, or stopping them) withou knowing or affecting the document's clip hierarchy. Storing clip references in variables also make our code more legible. You use a shorter, simpler variable name instead of a lengthy absolute or relative path through the movie clip hierarchy.

13.5.6.3 Using for-in to access movie clips

In Chapter 8, we saw how to enumerate an object's properties using a *for-in* loop. Recall that a *for-in* loop's iterator variable automatically cycles through all the properties of the object, so that the loop is executed once for each property:

```
for (var prop in someObject) {
  trace("the value of someObject." + prop + " is " + someObject[prop]);
}
```

Example 13-3 shows how to use a *for-in* loop to enumerate all the clips that reside on a given timeline.

Example 13-3. Finding movie clips on a timeline

```
for (var property in someClip) {
  // Check if the current property of someClip is a movie clip
  if (typeof someClip[property] =  = "movieclip") {
    trace("Found instance: " + someClip[property]._name);
    // Now do something to the clip
    someClip[property]._x = 300;
    someClip[property].play( );
  }
}
```

The *for-in* loop gives us convenient access to the clips contained by a specific clip instance or main movie. Using *for-in* , we

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

control any clip on any timeline, whether or not we know the clip's name and whether the clip was created manually or programmatically.

Example 13-4 shows a recursive version of Example 13-3. It finds all the clip instances on a timeline, plus the clip instances c nested timelines.

Example 13-4. Finding all movie clips on a timeline recursively

```
function findClips (theClip, indentSpaces) {
  // Use spaces to indent the child clips on each successive tier
  var indent = " ";
  for (var i = 0; i < indentSpaces; i++) {
    indent += " ";
  }
  for (var property in theClip) {
    // Check if the current property of theClip is a movie clip
    if (typeof theClip[property] =  = "movieclip") {
      trace(indent + theClip[property]._name);
      // Check if this clip is parent to any other clips
      findClips(theClip[property], indentSpaces + 4);
    }
  }
}
findClips(_root, 0); // Find all clip instances descended from main timeline
```

For more information on recursion, see Section 9.9 in Chapter 9.

13.5.6.4 The _name property

As we saw earlier in this chapter under Section 13.3.3, every instance's name is stored as a string in the built-in property _n We can use that property, as we saw in Example 13-1, to determine the name of the current clip or the name of some other cli an instance hierarchy:

```
this._name;         // The current instance's name
_parent._name       // The name of the clip that contains the current clip
```

The `_name` property comes in handy when we want to perform conditional operations on clips according to their identities. I example, here we duplicate the `seedClip` clip when it loads:

```
onClipEvent (load) {
  if (this._name =  = "seedClip") {
    this.duplicateMovieClip("clipCopy", 0);
  }
}
```

By checking explicitly for the `seedClip` name, we prevent infinite recursion—without our conditional statement, the *load* handler of each duplicated clip would cause the clip to duplicate itself.

13.5.6.5 The _target property

Every movie clip instance has a built-in `_target` property, which is a string that specifies the clip's absolute path using the

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

deprecated Flash 4 "slash" notation. For example, if `clipB` is placed inside `clipA` , and `clipA` is placed on the main tim the `_target` property of these clips is as follows:

```
_root._target                    // Contains: "/"
_root.clipA._target              // Contains: "/clipA"
_root.clipA.clipB._target        // Contains: "/clipA/clipB"
```

13.5.6.6 The targetPath( ) function

The *targetPath( )* function returns a string that contains the clip's absolute reference path, expressed using dot notation. The *targetPath( )* function is the modern, object-oriented equivalent of `_target` . It takes the form:

```
targetPath(theClip)
```

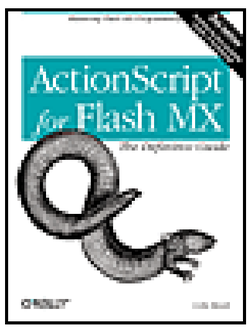
where *`theClip`* is the identifier of the clip whose absolute reference we wish to retrieve. Here are some examples, using o familiar example hierarchy:

```
targetPath(_root);               // Contains: "_level0"
targetPath(_root.clipA);         // Contains: "_level0.clipA"
targetPath(_root.clipA.clipB);   // Contains: "_level0.clipA.clipB"
```

The *targetPath( )* function gives us the complete path to a clip, whereas the `_name` property gives us only the name of the cli (This is analogous to having a complete file path versus just the filename.) So, we can use *targetPath( )* to compose code that controls clips based not only on their name but also on their location. For example, we might create a generic navigational bu that, by examining its *targetPath( )* , sets its own color to match the section of content within which it resides. See the exampl under the *Selection* object in the ActionScript Language Reference for a demonstration of *targetPath( )* in action.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir



ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Chapter 13. Movie Clips

13.6 Removing Clip Instances and Main Movies

We've seen how to create and refer to movie clips; now let's see how to turn them into so many recycled electrons (in other words, blow 'em away).

The manner in which we create an instance or a movie determines the technique we use to remove that instance or movie later. We can remove movies and instances explicitly using *unloadMovie( )* and *removeMovieClip( )* . Additionally, we can evict a clip implicitly by using *loadMovie( )* , *attachMovie( )* , or *duplicateMovieClip( )* to place a new clip in its stead. Let's look at these techniques individually.

13.6.1 Using unloadMovie( ) with Instances and Levels

The built-in *unloadMovie( )* function can remove any clip instance or main movie—both those created manually and those created via *loadMovie( )* , *duplicateMovieClip( )* , and *attachMovie( )* . It can be invoked either as a global function or as a instance-level method:

```
unloadMovie(clipOrLevel);   // Global function
clipOrLevel.unloadMovie( );  // Method
```

In the global function form of *unloadMovie( )* , `clipOrLevel` is a string indicating the path to the clip or level to unload. Because movie clips are converted to paths when used as strings, `clipOrLevel` can also be a movie clip reference. In the method form of *unloadMovie( )* , `clipOrLevel` must be a reference to a movie clip object. The exact behavior of *unloadMovie( )* varies according to whether it is used on a level or an instance.

13.6.1.1 Using unloadMovie( ) with levels

When applied to a level in the document stack (e.g., `_level0` , `_level1` , or `_level2`), *unloadMovie( )* completely removes the target level and the movie that the level contains. Subsequent references to the removed level yield `undefine` . Removing document levels is the most common use of the *unloadMovie( )* function:

```
unloadMovie("_level1");
_level1.unloadMovie();
```

13.6.1.2 Using unloadMovie( ) with instances

When applied to an instance (whether the instance is manually or programmatically created), *unloadMovie( )* removes the *contents* of the clip, *but it does not remove the clip itself* ! The timeline and canvas of the clip are removed, but an empty shel remains on stage. That shell can be referenced until the instance is removed permanently via *removeMovieClip( )* or until the span of frames on which the instance resides ends. Furthermore, any *onClipEvent( )* handlers on the shell remain active.

This "partial" deletion of the instance presents an interesting possibility; it lets us maintain a generic container clip whose contents can be changed repeatedly via *loadMovie( )* and *unloadMovie( )* . For example, we could use a single clip to load sections of a web site or images in a photo album. The following series of statements demonstrates the technique with an

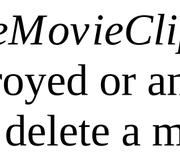
downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

instance called `clipA` (though in a real application, these statements would include the appropriate preloader code):

```
clipA.loadMovie("section1.swf");  // Load a document into clipA
clipA.unloadMovie( );              // Unload the document, leaving clipA intact
clipA.loadMovie("section2.swf");  // Load another document into clipA
```

One note of caution with this approach: when used on an instance, *unloadMovie( )* removes all custom properties of the clip contained by the instance. Physical properties — such as `_x` and `_alpha` — persist, but custom variables, functions, and event handler callbacks are lost.



If you use the global function form of *unloadMovie( )* with a nonexistent clip or level instance as its argument, the clip from which you invoked the *unloadMovie( )* function will, itself, unload.

For example, if `_level1` is undefined, and we issue the following code from the main timeline of `_level0` , then `_level0` will unload:

```
unloadMovie(_level1);
```

Yes, there's some logic to this behavior, but we'll cover that later in this chapter under Section 13.7. You can avoid the problem by using a string when specifying the *`clipOrLevel`* argument of *unloadMovie( )* or by checking explicitly that *`clipOrLevel`* exists before unloading it. Here's an example of each approach:

```
unloadMovie("_level1");  // clipOrLevel specified as a string
if (_level1) {           // Explicit check to make sure level exists
  unloadMovie(_level1);
}
```

13.6.2 Using removeMovieClip( ) to Delete Instances

To delete instances created via *duplicateMovieClip( )* , *attachMovie( )* or *createEmptyMovieClip( )* , we can use *removeMovieClip( )* . We delete an instance when it is no longer needed in our application, such as when an enemy spaceship is destroyed or an alert window is closed. Note that *removeMovieClip( )* works on duplicated or attached instances only. It cannot delete a manually created instance or a main movie. Like *unloadMovie( )* , *removeMovieClip( )* can be used in both method and global function form (though the syntax is different, the effect is the same):

```
removeMovieClip(theClip)    // Global function
theClip.removeMovieClip()   // Method
```

In the global function form of *removeMovieClip( )* , *`theClip`* is a string indicating the path to the clip to remove. Because movie clips are converted to paths when used as strings, *`theClip`* can also be a movie clip reference. In the method form of *removeMovieClip( )* , *`theClip`* must be a reference to a movie clip object.

Unlike using *unloadMovie( )* , deleting an instance via *removeMovieClip( )* completely obliterates the entire clip object, leaving no shell or trace of the clip and its properties. When we execute *theClip.removeMovieClip( )* , future references to *`theClip`* yield `undefined` .

13.6.3 Removing Manually Created Instances Manually

Clip instances created manually in the Flash authoring tool have a limited life span—they are removed when the playhead enters a keyframe that does not include them. Hence, manually created movie clips live in fear of the almighty blank keyframe.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Remember that when a movie clip disappears from the timeline, it ceases to exist as a data object. All variables, functions, methods, and properties that have been defined inside it are lost. Therefore, if we want a clip's information or functions to persist, we should be careful about removing the clip manually, and we should ensure that the span of frames on which the clip resides extends to the point where we need that clip's information. (In fact, to avoid this worry entirely, we should attach most permanent code to a frame in the main movie timeline or to the `_global` object.) To hide a clip while it's present on the timeline, simply set the clip's `_visible` property to `false` . Setting a clip's `_x` property to a very large positive number or very small negative number will also hide it from the user's view, but this approach is discouraged because the clip is still rendered off screen, consuming resources.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Chapter 13. Movie Clips

13.7 Method Versus Global Function Overlap Issues

As we've mentioned several times during this chapter, some movie clip methods have the same name as equivalent global functions. You can see this for yourself in the Flash authoring tool. Open the Actions panel, make sure you're in Expert Mode, and then take a look in the Actions folder under Movie Control and Movie Clip Control. You'll see a list of Actions, including *gotoAndPlay( )*, *gotoAndStop( )*, *nextFrame( )*, and *unloadMovie( )*. These Actions are also available as movie clip methods. The duplication is not purely a matter of categorization; the Actions are global functions, fully distinct from the corresponding movie clip methods.

So, when we execute:

```
theClip.gotoAndPlay(5);
```

we're accessing the *movie clip method* named *gotoAndPlay( )*. But when we execute:

```
gotoAndPlay(5);
```

we're accessing the *global function* called *gotoAndPlay( )*. These two commands have the same name, but they are not the same thing. The *gotoAndPlay( )* global function operates on the current instance or movie. The *gotoAndPlay( )* method operates on the clip object through which it is invoked. Most of the time, this subtle difference is of no consequence. But for some overlapping method/function pairs, this difference is potentially quite vexing.

Some global functions require a *target* parameter that specifies the clip on which the function should operate. This *target* parameter is not required by the comparable clip methods because the methods automatically operate on the clips through which they are invoked. For example, in its method form, *unloadMovie( )* works like this:

```
theClip.unloadMovie();
```

As a method, *unloadMovie( )* is invoked without parameters, and it automatically affects *theClip*. But in its global function form, *unloadMovie( )* works like this:

```
unloadMovie(target);
```

The global function version of *unloadMovie( )* requires *target* as a parameter that specifies which movie to unload. Why should this be a problem? Well, the first reason is that we may mistakenly expect to be able to unload the current document by using the global version of *unloadMovie( )* without any parameters, as we'd use *gotoAndPlay( )* without parameters:

```
unloadMovie();
```

This format does *not* unload the current clip. It causes a "Wrong number of parameters" error. The second reason that *target* parameters in global functions can cause problems is a little more complex and can be quite a pain to track down if you're not expecting it. To supply a *target* clip to a global function that requires a *target* parameter, we can use either a string, which expresses the path to the clip we wish to affect, or a clip reference. For example:

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

```
unloadMovie(_level1);    // Target clip is a reference
unloadMovie("_level1");  // Target clip is a string
```

We can use a reference simply because references to clip objects are converted to movie clip paths when used in a string context. This is simple enough, but if the `target` parameter resolves to an empty string or an `undefined` value, the *function operates on the current timeline*!

These examples demonstrate how an incorrect target clip reference can unintentionally unload the current timeline:

```
unloadMovie(x);   // If x doesn't exist, x yields undefined, so
                  // the function operates on the current timeline
unloadMovie("");  // The target is the empty string, so the function operates
                  // on the current timeline
```

This can cause some quite unexpected results. Consider what happens if we refer to a level that doesn't exist:

```
unloadMovie(_level1);
```

If `_level1` is empty, the interpreter resolves the reference as though it were an undeclared variable. This yields `undefined`, so the function operates on the current timeline, not `_level1`! So, how do we accommodate this behavior? There are a few options. We can check for the existence of our target before executing a function on it:

```
if (_level1) {
  unloadMovie(_level1);
}
```

Or, we can choose to always use a string to indicate the path to our target. If the path specified in our string does not resolve to a real clip, the function fails silently:

```
unloadMovie("_level1");
```

In some cases, we can use the equivalent numeric function for our operation:

```
unloadMovieNum(1);
```

Finally, we can choose to avoid the issue altogether by always using clip methods:

```
_level1.unloadMovie();
```

For reference, here are the troublemakers (the ActionScript global functions that take `target` parameters):

duplicateMovieClip( )
loadMovie( )
loadVariables( )
print( )
printAsBitmap( )
removeMovieClip( )
startDrag( )
unloadMovie( )

If you're experiencing unexplained problems in a movie, you may want to check this list to see if you're misusing a global function. When passing a clip reference as a `target` parameter, be sure to double-check your syntax.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Chapter 13. Movie Clips

13.8 Drawing in a Movie Clip at Runtime

Flash MX introduces the ability to draw strokes, curves, shapes, and fills in a movie clip using the *Drawing API*, a collection of methods for drawing at runtime. The Drawing API methods are documented in the ActionScript Language Reference, under the following entries:

MovieClip.beginFill( )
MovieClip.beginGradientFill( )
MovieClip.clear( )
MovieClip.curveTo( )
MovieClip.endFill( )
MovieClip.lineStyle( )
MovieClip.lineTo( )
MovieClip.moveTo( )

The *Drawing API* uses the concept of a *drawing pen* (or simply *pen*) to refer to the current drawing position, similar to the drawing pen used in old line plotters. Initially, the drawing pen resides at the registration point of a movie clip. Using the drawing methods, we can:

- Move the pen without drawing any lines or fills, via *MovieClip.moveTo( )*
- Draw a straight line from the pen's current position to a specific point, via *MovieClip.lineTo( )*
- Draw a curved line from the pen's current position to a specific point, via *MovieClip.curveTo( )*
- Draw a shape, via *MovieClip.beginFill( )* and *MovieClip.endFill( )* or *MovieClip.beginGradientFill( )* and *MovieClip.endFill( )*.

Before, during, or after drawing, we can specify the characteristics of the drawing stroke used in any drawing operation, via *MovieClip.lineStyle( )*. To remove a drawing, we use *MovieClip.clear( )*.

Notice that the Drawing API does not include methods for drawing shapes such as a triangle, rectangle, or circle. We must draw these using the primitive drawing methods, as demonstrated under *MovieClip.beginFill( )* in the *Language Reference*.

Detailed coverage of the drawing methods is left to the ActionScript Language Reference. For now, let's take a look at a couple of examples showing the general use of the Drawing API.

Example 13-5 creates a 100-pixel-long, 2-pixel-thick, straight line extending to the right from the current clip's registration point.

Example 13-5. Drawing a straight line

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

```
// Set stroke to 2-point
this.lineStyle(2);
// Draw the line
this.lineTo(100,0);
```

Example 13-6 creates a 200-pixel-wide, red square centered on the current clip's registration point.

Example 13-6. Drawing a square

```
// Set stroke to 3-point
this.lineStyle(3);
// Move the pen to 100 pixels left and above the registration point
this.moveTo(-100,-100);
// Start our red square shape
this.beginFill(0xFF0000);
// Draw the lines of our square
this.lineTo(100, -100);
this.lineTo(100, 100);
this.lineTo(-100, 100);
this.lineTo(-100, -100);
// Close our red square shape
this.endFill( );
```

For a variety of interesting applications of the Drawing API (including drawing arcs, polygons, dashed lines, stars, and wedges), see:

http://www.formequalsfunction.com/downloads/drawmethods.html

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Chapter 13. Movie Clips

13.9 Using Movie Clips as Buttons

As of Flash Player 6, movie clips (but not main movies) have all the features previously reserved for button symbols. A movie clip can dynamically define and redefine button event handlers, a *hit region*, and *Up*, *Over*, and *Down* states; and, unlike button symbols, movie clips can be instantiated dynamically at runtime. If you are implementing simple interactivity at authoring time, you can continue to use button symbols happily. But if you are generating complex, dynamic button behavior at runtime, you'll want to use movie clips.

The first step in implementing button-like behaviors for a movie clip is to define one or more button events for the clip. Normally, this means assigning a callback function to a predefined button event property (shown next), but the *on(`event`)* button syntax can also be applied to a movie clip directly in the authoring tool. The following code creates a movie clip, adds a text field to it, and then defines the *onRelease( )* button event handler:

```
// Create clip
this.createEmptyMovieClip("submit_mc", 1);

// Add text field
this.submit_mc.createTextField("submit_txt", 1, 0, 0, 50, 20);
this.submit_mc.submit_txt.text = "Submit";

// Define button handler
this.submit_mc.onRelease = function ( ) {
  trace("You pressed the submit button.");
}
```

As is, this movie clip operates perfectly well as a button. But suppose we want to make the word "Submit" easier for users to click. To define a larger *hit area* (the region that activates the button) for our movie clip, we'll create a separate *hit area movie clip* as follows:

1. Create a movie clip the size of the desired hit area.
2. Conceal the hit area movie clip by setting its `_visible` property to `false`.
3. Assign the hit area movie clip to the `hitArea` property of the clip with the button events.

The movie clip that acts as the hit area can reside on any timeline, but it is normally placed inside the clip with the button events so that the hit area moves and scales with its parent. For example, the following code uses the Drawing API to create a hit area movie clip larger than the word "Submit":

```
// Create hit_mc inside submit_mc
this.submit_mc.createEmptyMovieClip("hit_mc", 0);

// Draw a rectangle in hit_mc
this.submit_mc.hit_mc.moveTo(-30,-15);
this.submit_mc.hit_mc.beginFill(0xFF0000);
this.submit_mc.hit_mc.lineTo(80, -15);
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

```
this.submit_mc.hit_mc.lineTo(80, 35);
this.submit_mc.hit_mc.lineTo(-30, 35);
this.submit_mc.hit_mc.lineTo(-30, -15);
this.submit_mc.hit_mc.endFill( );

// Hide the hit area movie clip
this.submit_mc.hit_mc._visible = false;

// Set hit_mc as submit_mc's hit area
this.submit_mc.hitArea = this.submit_mc.hit_mc;
```

So far, our example does not change visually when the mouse is pressed or moved over the Submit button. Using ActionScript, visual changes to a button can be implemented dynamically within each button event handler. For example, we can bold the word "Submit" from our `submit_mc`'s *onRollOver( )* event as follows:

```
// Bold text on roll over
this.submit_mc.onRollOver = function ( ) {
  this.submit_txt.setTextFormat(new TextFormat(null, null, null, true));
}

// Normal text on roll out
this.submit_mc.onRollOut = function ( ) {
  this.submit_txt.setTextFormat(new TextFormat(null, null, null, false));
}
```

Alternatively, if we are creating our movie clip in the authoring tool, we can define button-state visual changes by creating keyframes with the special labels `_up`, `_over`, and `_down`, corresponding to the button states Up (mouse is not over the button), Over (mouse is over the button), and Down (button is being pressed). (Use the Property inspector to assign a label to a keyframe.) When the mouse interacts with the movie clip, Flash automatically moves the clip's playhead to the appropriately labeled frame. For example, when the mouse moves over the clip, Flash performs the equivalent of this:

```
theClip.gotoAndStop("_over");
```

At each of the labeled button-state frames, we can issue a *play( )* command to create animated button effects. However, we must be sure to also issue a *stop( )* command on frame 1 of the movie clip, which prevents the movie from playing through all its button states. Normally, this first frame is labeled `_up` (the default inactive state for the button).

Movie clips with button events can also use the button properties `enabled` (used to toggle button behavior on and off), `useHandCursor` (used to prevent or enable changes to the mouse pointer when it is over the button), and `trackAsMenu` (used to modify the button's *onRelease( )* handler requirements, enabling menu-style behavior). For example, we can suppress the display of the hand mouse pointer for our `submit_mc` clip as follows:

```
submit_mc.useHandCursor = false;
```

Typical web browser Submit buttons do not show a hand pointer on rollover. For deeper discussion of the button properties available to movie clips, see the *MovieClip* class in the ActionScript Language Reference.

Nested button behavior is not supported for movie clips. That is, movie clips inside a movie clip with button behavior cannot themselves define button behaviors. Furthermore, if a mask and a masked movie clip both define button handlers, only the mask movie clip's button handlers are activated.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

The various types of event handlers for buttons and movie clips have important scope differences, which are discussed in Chapter 10 under Section 10.9 and summarized in Table 10-1.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Chapter 13. Movie Clips

13.10 Input Focus and Movie Clips

As of Flash Player 6, movie clips instances (but not main movies) can receive keyboard input focus, allowing them to be controlled by keystrokes, much like a button object is controlled. Input focus is most commonly used by components that take input in complex graphical user interfaces. For example, a list box can allow the user to scroll through its items via the up and down arrow keys. The Enter key activates the selected item.

Generally speaking, if a keyboard-driven action is not tied directly to a specific movie clip, you should use the *Key* object's listener events — not movie clips with input focus — to trap the keystrokes. See *Key* in the *Language Reference*.

Despite what you might assume, movie clips do not receive keyboard input focus when they are clicked. Instead, they receive focus either due to the Tab key being pressed or, programmatically, by using the *setFocus( )* method. To allow a movie clip to receive input focus programmatically, we must explicitly set its `focusEnabled` property to `true`. In contrast, movie clips with button handlers can always receive input focus, even when `focusEnabled` is `false`. Once `focusEnabled` is `true` for a clip, we can focus the clip programmatically with *Selection.setFocus( )*. The user can also focus the clip with the Tab key, provided that the movie clip's `tabEnabled` property is `true`. The movie clip's position in the tab order is dictated by its `tabIndex` property.

When a movie clip has input focus, its *onKeyUp( )* and *onKeyDown( )* event handlers become active in their callback-function form. From these handlers, we can implement keyboard-specific behaviors, such as expanding a hierarchical menu when the right arrow key is pressed or jumping to the nearest item in a list when a letter is pressed.

We can detect when a movie clip gains and loses input focus via the *onSetFocus( )* and *onKillFocus( )* events, defined both by the *MovieClip* class and the *Selection* object.

By default, keyboard focus is indicated by a yellow rectangle. The highlight is useful for debugging the tab order, but it may not be visually appealing. To remove the yellow rectangle, set the `MovieClip._focusrect` property to `false`. If you do this, you should give the end user some indicator as to which screen element has keyboard focus at runtime. To do so, use the clip's *onSetFocus( )* and *onKillfocus( )* handlers, as shown in the next example, to toggle some indicator of focus.

> In Flash Player 6, moving the mouse while a movie clip or button has focus removes focus from that clip or button.

The following simplified code shows how an item in a shopping basket might respond to being focused and deleted via the *X* key. In a real application, the behavior would most likely be implemented at the class level rather than on a specific movie clip instance. The *doHighlight( )* and *doRemoveHighlight( )* methods are left as an exercise for the reader. You might use those handlers to jump the playhead to a label that shows or hides some focus indicator, such as the dotted line or thick border typically seen around the button with focus in Windows or Macintosh dialog boxes.

downloaded from: lib.ommolketab.ir

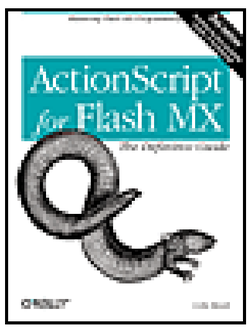
downloaded from: lib.ommolketab.ir

```
// Allow user to focus item_mc via Tab key
item_mc.tabEnabled = true;

// Allow Selection.setFocus( ) to focus item_mc
item_mc.focusEnabled = true;

// Highlight item_mc when it is focused
item_mc.onSetFocus = function ( ) {
  this.doHighlight( );
}

// Remove highlight from item_mc when it loses focus
item_mc.onKillFocus = function ( ) {
  this.doRemoveHighlight( );
}

// Remove item_mc when "x" is pressed and item_mc has focus
item_mc.onKeyDown = function ( ) {
  if (Key.getCode( ) =  = 88) {  // The keycode for "x" is 88
    this.removeMovieClip( );
  }
}
```

For much more information on movie clip input focus, consult the ActionScript Language Reference entries for the properties and methods discussed in this section.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir



ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Chapter 13. Movie Clips

13.11 Building a Clock with Clips

Now that you've learned the fundamentals of movie clip programming, let's put this knowledge to use by creating a sample ar clock application, which exemplifies the typical role of movie clips as basic content containers. See also the various versions multiple-choice quiz posted at the online Code Depot.

In this chapter we saw how to create movie clips with *attachMovie( )* and how to set movie clip properties with the dot operat With these relatively simple tools and a little help from the *Date* and *Color* classes, we have everything we need to make a clc with functional hour, minute, and second hands.

First, we'll make the face and hands of the clock using the following steps (notice that we don't place the parts of our clock on main Stage—our clock will be generated entirely through ActionScript):

1. Create a new, empty Flash movie.
2. Create a movie clip symbol, named *clockFace* , that contains a 100-pixel-wide black circle shape, centered on the clip's registration point.
3. Create a movie clip symbol, named *hand* , that contains a 50-pixel-long, vertical red line.
4. Select the line in *hand* ; then choose Window Info (Flash MX) or Window Panels Info (Flash 5).
5. Because we want the hands to rotate around the center of the clock, we must position the line so that one end is at the registration point (the center) of the hand clip. Therefore, position the bottom of the line at the center of the clip by setti the line's x-coordinate to 0 and its y-coordinate to -50.

Now we have to export our *clockFace* and *hand* symbols, so that instances of them can be attached dynamically to our movie:

1. In the Library, select the *clockFace* clip; then select Linkage from the pop-up Options menu. The Linkage Properties di box appears.
2. Select the Export For ActionScript checkbox.
3. In the Identifier box, type **`clockFace`** , and then click OK.
4. Repeat Steps 1 through 3 to export the *hand* clip, giving it the identifier **`hand`** .

The face and hands of our clock are complete and ready to be attached to our movie. Now let's write the script that places the assets on stage and refreshes them with each passing second:

1. Add the script shown in Example 13-7 to frame 1 of *Layer 1* of the main timeline.
2. Rename *Layer 1* to *scripts* .

Skim Example 13-7 in its entirety first; then we'll dissect it.

Example 13-7. An analog clock

```
// Create clock face and hands
attachMovie("clockFace", "clockFace_mc", 0);
attachMovie("hand", "secondHand_mc", 3);
attachMovie("hand", "minuteHand_mc", 2);
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

```
attachMovie("hand", "hourHand_mc", 1);

// Position and size the clock face
clockFace_mc._x = 275;
clockFace_mc._y = 200;
clockFace_mc._height = 150;
clockFace_mc._width = 150;

// Position, size, and color the clock hands
secondHand_mc._x = clockFace_mc._x;
secondHand_mc._y = clockFace_mc._y;
secondHand_mc._height = clockFace_mc._height / 2.2;
secondHandColor = new Color(secondHand_mc);
secondHandColor.setRGB(0xFFFFFF);
minuteHand_mc._x = clockFace_mc._x;
minuteHand_mc._y = clockFace_mc._y;
minuteHand_mc._height = clockFace_mc._height / 2.5;
hourHand_mc._x = clockFace_mc._x;
hourHand_mc._y = clockFace_mc._y;
hourHand_mc._height = clockFace_mc._height / 3.5;

// Update the rotation of hands with each passing frame
function updateClock () {
  var now = new Date( );
  var dayPercent = (now.getHours( ) > 12 ?
                    now.getHours( ) - 12 : now.getHours( )) / 12;
  var hourPercent = now.getMinutes( )/60;
  var minutePercent = now.getSeconds( )/60;
  hourHand_mc._rotation = 360 * dayPercent + hourPercent * (360 / 12);
  minuteHand_mc._rotation = 360 * hourPercent;
  secondHand_mc._rotation = 360 * minutePercent;
}

// Update the clock every 100 milliseconds
setInterval(updateClock, 100);
```

That's a lot of code, so let's review it.

We first attach the `clockFace` clip and assign it a depth of 0 (we want it to appear behind our clock's hands):

```
attachMovie("clockFace", "clockFace_mc", 0);
```

Next, we attach three instances of the *hand* symbol, assigning them the names `secondHand_mc` , `minuteHand_mc` , ar `hourHand_mc` . Each hand resides on its own layer in the programmatically generated content stack above the main timeli The `secondHand_mc` clip (depth 3) sits on top of the `minuteHand_mc` clip (depth 2), which sits on top of the `hourHand_mc` clip (depth 1):

```
attachMovie("hand", "secondHand_mc", 3);
attachMovie("hand", "minuteHand_mc", 2);
attachMovie("hand", "hourHand_mc", 1);
```

We want our clock centered — not in the top-left corner of the Stage — so, we center the `clockFace_mc` clip on stage an make it larger using the `_height` and `_width` properties. In this example, we assume the movie size is the default (550 x

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

pixels), but we could have used `Stage.width` and `Stage.height` to dynamically retrieve the dimensions of the movi runtime. As an exercise, use *Stage.onResize( )* to keep the clock centered even when the movie is resized.

```
clockFace_mc._x = 275;
clockFace_mc._y = 200;
clockFace_mc._height = 150;
clockFace_mc._width = 150;
```

Next, we move the `secondHand_mc` clip onto the clock and make it almost as long as the radius of the `clockFace_mc`

```
secondHand_mc._x = clockFace_mc._x;
secondHand_mc._y = clockFace_mc._y;
secondHand_mc._height = clockFace_mc._height / 2.2;
```

Remember that the line in the *hand* symbol is red, so all our *hand* instances thus far are also red. To make our `secondHand` clip stand out, we color it white using the *Color* class. Note the use of the hexadecimal color value `0xFFFFFF` (see the *Colo* Class in the *Language Reference* for more information on manipulating color):

```
// Create a new Color object to control secondHand_mc
secondHandColor = new Color(secondHand_mc);
// Assign secondHand_mc the color white
secondHandColor.setRGB(0xFFFFFF);
```

Next, we set the position and size of the `minuteHand_mc` and `hourHand_mc` clips, just as we did for the `secondHand_mc` clip:

```
// Place minuteHand_mc on top of clockFace_mc
minuteHand_mc._x = clockFace_mc._x;
minuteHand_mc._y = clockFace_mc._y;
// Make minuteHand_mc shorter than secondHand_mc
minuteHand_mc._height = clockFace_mc._height / 2.5;
// Place hourHand_mc on top of clockFace_mc
hourHand_mc._x = clockFace_mc._x;
hourHand_mc._y = clockFace_mc._y;
// Make hourHand_mc the shortest of all
hourHand_mc._height = clockFace_mc._height / 3.5;
```

Now we have to set the rotation of the hands on the clock to reflect the current time. However, we don't just want to set the rotation once. We want to set it repeatedly, so that our clock hands animate over time. Therefore, we put our rotation code in a function called *updateClock( )* , which we'll call periodically:

```
function updateClock ( ) {
  // Store the current time in now
  var now = new Date( );

  // getHours( ) works on a 24-hour clock. If the current hour is greater
  // than 12, we subtract 12 to convert to a regular 12-hour clock.
  var dayPercent = (now.getHours( ) > 12 ?
                    now.getHours( ) - 12 : now.getHours( )) / 12;

  // Determine how many minutes of the current hour have passed, as a percenta
  var hourPercent = now.getMinutes( )/60;
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

```
  // Determine how many seconds of the current minute have passed, as a percen
  var minutePercent = now.getSeconds( )/60;

  // Rotate the hands by the appropriate amount around the clock
  hourHand_mc._rotation = 360 * dayPercent + hourPercent * (360 / 12);
  minuteHand_mc._rotation = 360 * hourPercent;
  secondHand_mc._rotation = 360 * minutePercent;
}
```

The first task of *updateClock( )* is to retrieve and store the current time. This is done by creating an instance of the *Date* class placing it in the local variable `now` . Next we determine, as a percentage, how far around the clock each hand should be placed—much like determining where to slice a pie. The current hour always represents some portion of 12, while the current minute and second always represent some portion of 60. We assign the `_rotation` of each hand based on those percentag We calculate the `hourHand_mc` clip's position to reflect not only the percent of the day but also the percent of the current h

Our clock is essentially finished. All that's left to do is call the *updateClock( )* function with the following line of code, which refreshes the clock display every 100 milliseconds:

```
setInterval(updateClock, 100);
```

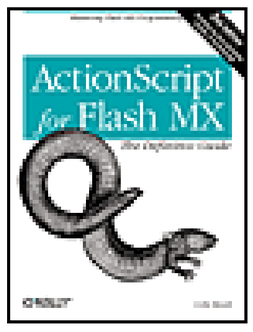
Test the movie to see if your clock works. If it doesn't, compare it to the sample clock *.fla* file provided at the online Code De or check your code against Example 13-7. Updating the clock 10 times per second should create a nice smooth sweep of the second hand, but it may also be overkill (or a drain on performance of a larger piece).

Reader Exercise: Modify Example 13-7 to update the clock only once per second, and add a "tick-tock" sound synchronized v the second hand's movement (which should give the illusion of a nice firm snap).

Think of other ways to expand on the clock application: Can you draw the clock programmatically with the Drawing API? Ca you make the clock more portable by turning it into a component? How about dynamically adding minute and hour markings the `clockFace_mc` clip? Can you modify it to display different or multiple time zones? Can you create a stopwatch, comp with a reset button?

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir



ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Chapter 13. Movie Clips

13.12 Onward!

We've come so far that there's not much more to move on to! Once you understand objects and movie clips thoroughly, you can tackle most ActionScript projects on your own. But there's still some interesting ground ahead. Chapter 14 shows how to create *MovieClip* subclasses and components, such as the UI Components shipped with Flash MX.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Part I: ActionScript Fundamentals

Chapter 14. Movie Clip Subclasses and Components

In Chapter 12 we created a custom *Ball* class and used it to instantiate individual *Ball* objects. When it came time to display the ball objects, we were faced with several options. A separate *Room* class might contain balls and be responsible for drawing them on screen with the Drawing API. Or, a *Ball* object might have a property, `mc` , that contains a movie clip used to represent the ball. In some cases, it's redundant to have one object that represents a thing (such as a ball) and a separate movie clip object that displays it on screen. Movie clips, themselves, are already objects, so it often makes sense to use a movie clip to both describe an object and display that object on screen. In Flash MX, we can do so by creating a subclass based on the *MovieClip* class, thus creating a specialized type of object that provides built-in screen display.

In Flash 5, creating a *MovieClip* subclass involved an arduous series of workarounds that proved impractical for most developers. Flash MX introduces a formal *MovieClip* subclassing architecture that involves the following new features: `#initclip` , *new MovieClip( )* , *Object.registerClass( )* , and *attachMovie( )* 's `initObj` parameter.

To see how *MovieClip* subclasses are created, we'll implement our *Ball* class from Chapter 12 as a *MovieClip* subclass. In our example, each ball instance will be a movie clip responsible for moving itself around the screen. Read through the code in Example 14-1. We'll discuss it in the sections that follow. Generally, the code uses the OOP techniques we studied in Chapter 12, but with a few twists required by the *MovieClip* subclassing architecture. All of the example code in this chapter can be downloaded from the online Code Depot.

Example 14-1. The Ball class, a sample MovieClip subclass

```
// =  =  =  =  =  =  =  =  =  =  =  =  =  =  =  =  =  =  =  =  =  =  =  =  =  =  =  =
// Create namespace in which to store classes
// =  =  =  =  =  =  =  =  =  =  =  =  =  =  =  =  =  =  =  =  =  =  =  =  =  =  =  =
if (_global.org =  = undefined) {
  _global.org = new Object( );
}
if (_global.org.moock =  = undefined) {
  _global.org.moock = new Object( );
}
// =  =  =  =  =  =  =  =  =  =  =  =  =  =  =  =  =  =  =  =  =  =  =  =  =  =  =  =
// Create the Ball class
// =  =  =  =  =  =  =  =  =  =  =  =  =  =  =  =  =  =  =  =  =  =  =  =  =  =  =  =
/*
 * Ball Class. Extends MovieClip.
 *   Version: 1.0.0
 *   Desc: A movie clip subclass for depicting balls.
 *         Requires a Library symbol named ballSymbol.
 *
 * Constructor Params:
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

```
 *   radius          -The ball's size (half its diameter)
 *   ballColor       -The ball's color
 *   x               -The ball's initial x position
 *   y               -The ball's initial y position
 *   velocity        -The ball's movement vector: {x:0, y:0}
 *
 * Methods:
 *   setSize( )       -Make the ball clip's size match the supplied radius
 *   setPosition( )   -Place the ball clip at an (x,y) coordinate
 *   setColor( )      -Apply an RGB color value to the ball clip
 *   move( )          -Apply the ball's velocity vector to its current positio
 *
 * Event Handlers:
 *   onEnterFrame( )  -Invokes move( ) every frame.
 */
/*
 * Ball Class Constructor (parameters are passed by attachMovie( )'s initObj)
 */
org.moock.Ball = function ( ) {
  // Create instance properties
  this.velocity = this.params.velocity;
  // Initialize instance
  this.setPosition(this.params.x, this.params.y);
  this.setColor(this.params.ballColor);
  this.setSize(this.params.radius);
  this.onEnterFrame = this.move;
  // Remove constructor params object from the instance
  delete this.params;
}
/*
 * Set MovieClip as Ball's superclass
 */
org.moock.Ball.prototype = new MovieClip( );
/*
 * Associate the Library's ballSymbol with the Ball class
 */
Object.registerClass("ballSymbol", org.moock.Ball);
/*
 * Instance Methods
 */
/*
 * Method: Ball.setSize( )
 *   Desc: Sets the ball clip's size
 *
 * Params:
 *   newRadius     -The ball's new radius
 */
org.moock.Ball.prototype.setSize = function (newRadius) {
  this._width = this._height = (newRadius * 2);
};
/*
 *  Method: Ball.setPosition( )
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

```
 *   Desc: Sets the ball clip's position to the point (x, y)
 *
 * Params:
 *   x             -The new horizontal position
 *   y             -The new vertical position
 */
org.moock.Ball.prototype.setPosition = function (x, y) {
  this._x = x;
  this._y = y;
};
/*
 * Method: Ball.setColor( )
 *   Desc: Changes the ball clip's color
 *
 * Params:
 *   newColor      -The numeric RGB color to apply to the ball clip
 *
 */
org.moock.Ball.prototype.setColor = function (newColor) {
  var col = new Color(this);
  col.setRGB(newColor);
};
/*
 * Method: Ball.move( )
 *   Desc: Moves the ball according to its velocity vector
 *
 * Params: None
 *
 */
org.moock.Ball.prototype.move = function ( ) {
  this._x += this.velocity.x;
  this._y += this.velocity.y;
}
// =  =  =  =  =  =  =  =  =  =  =  =  =  =  =  =  =  =  =  =  =  =  =
// START APPLICATION
// =  =  =  =  =  =  =  =  =  =  =  =  =  =  =  =  =  =  =  =  =  =  =
main( );
// Function: main( )
//     Desc: Application entry point
function main ( ) {
  // Create an object whose properties will be used
  // as the Ball constructor's parameters
  var ballParams = new Object( );
  ballParams.radius = 10;
  ballParams.ballColor = 0xFF0000;
  ballParams.x = 250;
  ballParams.y = 250;
  ballParams.velocity = {x:12, y:4};

  // Assign the ballParams object to a single property
  // of initObj, which we'll pass to attachMovie( ).
  // Properties of initObj are copied to each ball instance at attach-time.
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

```
    var initObj = new Object( );
    initObj.params = ballParams;

    // Create a ball
    this.attachMovie("ballSymbol", "redBall", 1, initObj);
}
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Chapter 14. Movie Clip Subclasses and Components

14.1 Creating the Library Symbol

Before returning to the code in Example 14-1, let's set up our movie's assets. Each movie clip subclass is associated with a movie clip symbol stored in the Library. To create instances of the subclass programmatically, we must export the symbol for use with ActionScript. Here are the steps we follow to create and export a movie clip symbol for our *Ball* class:

1. Create a movie clip symbol, named *ballSymbol*, that contains a circle shape.
2. In the Library, select *ballSymbol*.
3. From the Library's pop-up Options menu, select Linkage. The Linkage Properties dialog box appears.
4. Select the Export For ActionScript checkbox.
5. In the Identifier field, type `ballSymbol`.
6. Click OK.

Next, we'll create the *Ball* constructor function that will be invoked automatically every time an instance of *ballSymbol* is created.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Chapter 14. Movie Clip Subclasses and Components

14.2 Creating and Invoking the Subclass Constructor

For now, our *Ball* constructor (from Example 14-1) is defined on the main timeline of our *.fla* file. Later, we'll see how to move the constructor to our *ballSymbol* Library symbol. The *Ball* constructor looks very much like the constructor functions we saw in Chapter 12, with the added feature that it is defined in a global namespace corresponding to the domain name, `moock.org` (your classes should be defined in your domain's namespace, as in `com.yoursite.YourClass`):

```
/*
 * Ball Class Constructor.
 */
org.moock.Ball = function ( ) {
  // Create Instance properties.
  // ...
  // Initialize instance.
  // ...
};
```

However, you'll notice that our class constructor does not define any parameters! Remember that we're creating a *MovieClip* subclass, and movie clip instances are not created with the standard *new Object( )* instantiation, which facilitates parameters. Instead, movie clip instances are created with the functions *attachMovie( )* or *duplicateMovieClip( )*, which generate the instance and invoke its class constructor (if one is defined). Hence, to supply arguments to our *Ball* constructor function, we have to pass them through either *attachMovie( )* or *duplicateMovieClip( )*. But we can't pass the arguments directly, as we would with the syntax: *new Object(`arg1`,...`argn`)*. Instead, we must use a so-called *initObject*—an object whose properties are copied to the new instance just before the constructor function runs.

For example, the following code attaches an instance of the movie clip symbol exported as "`boxSymbol`":

```
var initObj = new Object( );
initObj.description = "A four-sided figure";
this.attachMovie("boxSymbol", "box_mc", 2, initObj);
```

The code names the new instance "`box_mc`", places it on depth 2, and passes it an initObject with a single property, `description`. After the code runs, the `box_mc` instance will have a `description` property set to the value "A four-sided figure".

In place of true parameters, *MovieClip* subclass constructors are expected to use the properties set by the *attachMovie( )* or *duplicateMovieClip( )* initObject. For example, our *Ball* constructor (shown next) relies on the initObject's `params` property—which itself is an object—to pass the parameters `x`, `y`, `velocity`, `radius`, and `ballColor` to the constructor:

```
org.moock.Ball = function ( ) {
  // Create instance properties
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

```
    this.velocity = this.params.velocity;

    // Initialize instance
    this.setPosition(this.params.x, this.params.y);
    this.setColor(this.params.ballColor);
    this.setSize(this.params.radius);
    this.onEnterFrame = this.move;

    // Remove constructor params object from the instance
    delete this.params;
}
```

When attaching a new instance, we create our initObject like this:

```
// Create an object whose properties will be used
// as the Ball constructor's parameters.
var ballParams = new Object( );
ballParams.radius = 10;
ballParams.ballColor = 0xFF0000;
ballParams.x = 250;
ballParams.y = 250;
ballParams.velocity = {x:12, y:4};

// Assign the ballParams object to the single initObj.params property.
var initObj = new Object( );
initObj.params = ballParams;

// Now pass initObj to the instance via attachMovie( ). The params
// property is copied to the new instance, with all the constructor
// parameters defined by ballParams.
this.attachMovie("ballSymbol", "redBall", 1, initObj);
```

In Example 14-1, these instance-creation statements occur within the *main( )* function, so both `ballParams` and `initObj` variables are local, and they are deleted automatically when *main( )* finishes executing. Otherwise, we would have had to delete `ballParams` and `initObj` explicitly, or supply our initObject as an object literal that would not persist after being passed to *attachMovie( )*.

You might wonder why we bothered with a `params` property. That is, why not just define the `radius`, `velocity`, `ballColor`, `_x`, and `_y` properties directly on the initObject and use them directly in our *Ball* constructor, as follows?

```
// Constructor code
org.moock.Ball = function ( ) {
  // Initialize instance.
  this.setColor(this.ballColor);
  this.setSize(this.radius);
  this.onEnterFrame = this.move;
};

// Attach movie call
var initObj = new Object( );
initObj.radius = 10;
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

```
initObj.velocity = {x:44, y:2};
initObj.ballColor = 0xFF0000;
initObj._x = 30;
initObj._y = 12;
this.attachMovie("ballSymbol", "redBall", 1, initObj);
```

While setting object properties directly on the `initObj` works, it poses two problems:

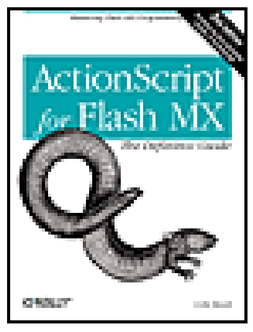
- Code is not properly centralized: the *attachMovie( )* function takes on the responsibility of creating instance properties, which should be the role of the constructor function.
- Constructor parameters (which should be temporary) are left as permanent properties of the new instance. The property `ballColor`, for example, is no longer needed after construction, yet it remains defined for as long as the instance exists.

To centralize our code and ensure that our instance is not unnecessarily cluttered with constructor-parameter properties, we define all our parameters on a single `params` property of the initObject. When the constructor finishes executing, it then deletes the `params` property:

```
// Remove constructor params object from the instance.
delete this.params;
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir



ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Chapter 14. Movie Clip Subclasses and Components

14.3 Assigning the MovieClip Superclass

We've created our *Ball* class constructor, and we have seen how to instantiate *Ball* objects with *attachMovie( )*. Now we have to make our *Ball* class a subclass of the *MovieClip* class. As we saw in Chapter 12, we declare a class's superclass like this:

```
SubClass.prototype = new SuperClass( );
```

Hence, it comes as no surprise that we declare *MovieClip* as *Ball*'s superclass like this:

```
/*
 * Set MovieClip as Ball's superclass.
 */
org.moock.Ball.prototype = new MovieClip( );
```

However, although we've defined *Ball* as a subclass of *MovieClip*, we still need to tell the interpreter to execute the *Ball* constructor when we create a new instance of our exported *ballSymbol* from the Library. To associate an exported symbol with the desired class constructor, we use *Object.registerClass( )*. Here we associate *ballSymbol* with the *Ball* class:

```
/*
 * Associate the Library's ballSymbol with the Ball class.
 */
Object.registerClass("ballSymbol", org.moock.Ball);
```

In plain English, this tells the interpreter, "when a *ballSymbol* instance is created, invoke the *org.moock.Ball( )* constructor, with the `this` keyword pointing to the new *ballSymbol* instance."

> The *new MovieClip( )* statement only establishes a class as a subclass of the *MovieClip* class (i.e., it merely establishes inheritance). It *cannot* be used to create new movie clips for use in a movie! Use *attachMovie( )*, *duplicateMovieClip( )*, or *createEmptyMovieClip( )* to create new instances of movie clips or movie clip subclasses, such as *Ball*.

For more information on *Object.registerClass( )*, see the ActionScript Language Reference.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Chapter 14. Movie Clip Subclasses and Components

14.4 Packaging Subclass Code and Library Symbols Together

With our superclass assigned, we've created a fully functioning *MovieClip* subclass, *Ball* . We can add methods and properties to it via `Ball.prototype` , just as we would with any class (as shown in Example 14-1). However, suppose we wanted to share our *Ball* class with another developer. We might sensibly put our class in an external *.as* file and tell the developer to `#include` the file. But we'd also have to tell her to make a Library symbol for the class and export that symbc as "`ballSymbol` ". If the developer doesn't export the symbol, or if she exports it with the wrong linkage identifier, our class won't work.

Maintaining both a class file and an accompanying movie clip symbol can be inconvenient and error prone. To make a *MovieClip* subclass more self-contained, we can place our code inside the movie clip symbol itself, between the `#initclip` and `#endinitclip` pragmas (creating a so-called *#initclip block*).

For example, to package our *Ball* class inside our *ballSymbol* movie clip, we place the following code on frame 1 of the *ballSymbol* 's timeline. (The function body code and some comments have been omitted for brevity).

```
#initclip

// =  =  =  =  =  =  =  =  =  =  =  =  =  =  =  =  =  =  =  =  =  =  =  =  =  =  =
// Create namespace in which to store classes
// =  =  =  =  =  =  =  =  =  =  =  =  =  =  =  =  =  =  =  =  =  =  =  =  =  =  =
if (_global.org =  = undefined) {
  _global.org = new Object( );
}
if (_global.org.moock =  = undefined) {
  _global.org.moock = new Object( );
}

// =  =  =  =  =  =  =  =  =  =  =  =  =  =  =  =  =  =  =  =  =  =  =  =  =  =  =
// Create the Ball class
// =  =  =  =  =  =  =  =  =  =  =  =  =  =  =  =  =  =  =  =  =  =  =  =  =  =  =

// Class Constructor
org.moock.Ball = function ( ) { ... };

// Set MovieClip as Ball's superclass
org.moock.Ball.prototype = new MovieClip( );

// Associate the Library's ballSymbol with the Ball class
Object.registerClass("ballSymbol", org.moock.Ball);

// Method: Ball.setSize( )
org.moock.Ball.prototype.setSize = function (newRadius) { ... };
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

```
// Method: Ball.setPosition( )
org.moock.Ball.prototype.setPosition = function (x, y) { ... };

// Method: Ball.setColor( )
org.moock.Ball.prototype.setColor = function (newColor) { ... };

// Method: Ball.move( )
org.moock.Ball.prototype.move = function ( ) { ... };

#endinitclip
```

An `#initclip` block can appear only on frame 1 of a movie clip timeline. If the code is placed on the main timeline or outside of frame 1 of a movie clip timeline, it generates a compile-time error. Code within the block is executed once, and only once, per movie clip symbol, just before the first instance of the symbol is created. By defining our *Ball* class inside the *ballSymbol* 's `#initclip` block, we guarantee that the *Ball* class constructor will be available before the first instance of *ballSymbol* appears. Methods and properties defined by *Ball* will be available immediately to each new *Ball* instance.

With our *Ball* class stored in our *ballSymbol* 's `#initclip` block, we can easily move the entire package between movies by dragging the *ballSymbol* from one *.fla* file's Library to another. Without any more effort, the destination *.fla* file can then attach and use instances of *ballSymbol* .

For more on `#initclip` , see the *Language Reference* .

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Chapter 14. Movie Clip Subclasses and Components

14.5 Making Movie Clip Components

So far, we've ignored the possibility that an instance of our *ballSymbol* may be dragged to the Stage manually in the Flash authoring environment. If that happens, the *Ball( )* constructor executes at runtime, even for instances dragged to the Stage during authoring (assuming that *Ball* has been defined as a subclass before the instance is placed on the timeline). But there's no opportunity to set parameters via an initObject—as with *attachMovie( )* or *duplicateMovieClip( )*—when the instance is created manually during authoring. To overcome this problem, we must define our movie clip symbol as a component.

A *component* is simply a movie clip symbol that provides a GUI for setting properties in author-time instances. Typically, components are also *MovieClip* subclasses that use the author-time properties as constructor parameters.

Flash MX components are the new and improved incarnation of Flash 5's Smart Clips.

We'll study component creation more closely in Chapter 16. For now, let's see how to turn our *ballSymbol* into a component. We'll continue working with *ballSymbol* from the last section, which defines the *Ball* class constructor within its `#initclip` block.

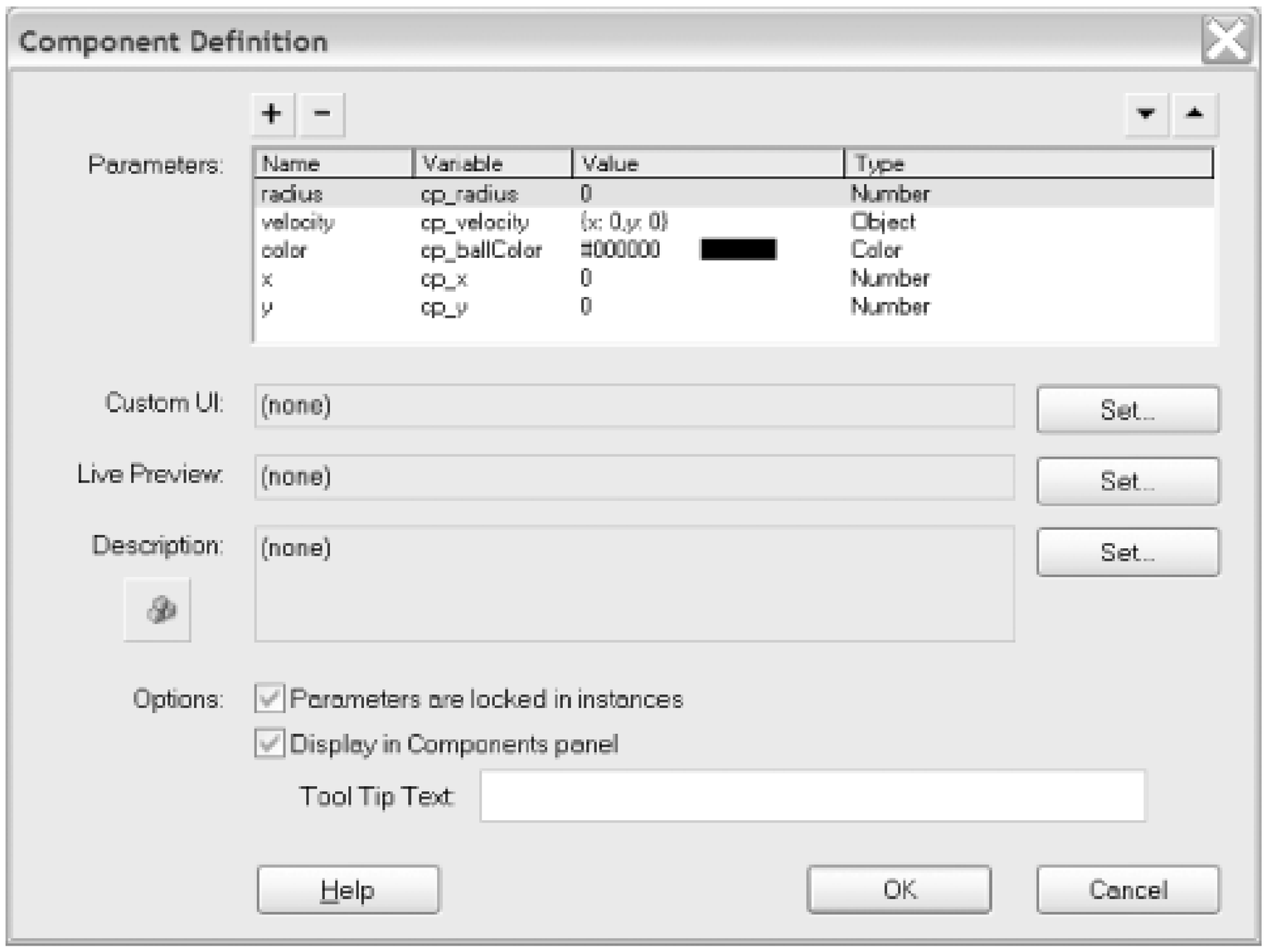
1. Select *ballSymbol* in the Library.
2. In the Library's pop-up Options menu, select Component Definition. The Component Definition dialog box appears.
3. Click the + button five times to insert placeholders for five new component parameters.
4. For the five parameters, under Name, enter **`radius`**, **`velocity`**, **`color`**, **`x`**, and **`y`**. These are the cosmetic author-time parameter names shown to the end user. You'll recognize them as our *Ball* constructor parameters.
5. For the five parameters, under Variable, enter **`cp_radius`**, **`cp_velocity`**, **`cp_ballColor`**, **`cp_x`**, and **`cp_y`**. These are the property names that will be set in each *ballSymbol* instance. The prefix `cp_` is our own convention, meaning "component parameter." We'll use it to identify and delete component parameters after our constructor is finished using them.
6. For the five parameters, using the pop-up menu under Type, choose Number, Object, Color, Number, and Number. These indicate the datatype of each parameter. The Color type also provides the user with a color picker for color selection.
7. Finally, for the five parameters, under Value, enter **`0`**, **`{x:0, y:0}`**, **`#000000`**, **`0`**, and **`0`**. Notice that for `velocity` and `color`, a specialized interface offers easy entry of object properties and color values. These are the default values, but the user can change them when he drags the instance to the Stage.
8. Click OK.

Figure 14-1 shows our *ballSymbol* component definition.

Figure 14-1. The ballSymbol component definition

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir



Henceforth, when a user drags an instance of *ballSymbol* to the Stage, the Property inspector will provide them with a GUI for setting the instance's "component parameters," as shown in Figure 14-2.

Figure 14-2. The ballSymbol component parameter GUI

The parameter values specified by the user in the Property inspector are automatically assigned to properties in the *ballSymbol* instance (the property names used are those we specified earlier in Step 5— `cp_radius`, `cp_velocity`, `cp_ballColor`, `cp_x`, and `cp_y`). Although we could simply use these instance properties as-is, we'll modify our *Ball* constructor to improve code modularity.

Recall that our original *Ball* constructor function expects its parameters to be supplied in a single package—the `params` object. Therefore, the constructor accesses each parameter as `this.params.`*`paramName`*, as shown in Example 14-1. The `params` object also neatly contains all our constructor parameters so that they can be deleted conveniently after construction. But our component parameters (`cp_radius`, `cp_velocity`, etc.), in contrast, are defined directly as properties of new *ballSymbol* instances, not on a `params` object. Just as with the *attachMovie( )* initObject, it's legal to use these direct properties as-is, without any constructor function intervention. However, doing so encroaches on the duties of our constructor. A normal constructor function is responsible for creating instance properties and performing instance initialization; it also uses constructor parameters only to initialize the new instance.

In addition, normal constructor parameters are local to the constructor function, and they expire automatically when the

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

constructor finishes executing. Our *Ball* constructor forces the component parameters (`cp_radius`, `cp_velocity`, etc.) into the role of constructor parameters by using them only to initialize the new *Ball* instance and by deleting them explicitly at the end of the function. In our *Ball* constructor, the component parameters are temporary and do not equate to the instance's internal properties. For example, we might ask the user to specify the ball's diameter as a component parameter but derive and store the ball's radius as a property instead. Distinguishing between component parameters and our *Ball* class's properties keeps our code centralized; properties are created in the constructor as expected, rather than through the Component Definition GUI.

Here's the revised code for our "componentized" *Ball* constructor. All other code for the *Ball* class remains the same. Notice how important code comments become—the constructor defines no parameters, so we must rely on comments to determine the expected parameters. (In the following code, the comments for version information, methods, and event handlers are omitted for brevity.) The constructor recognizes component parameters for deletion by looking for the `cp_` prefix, which we specified earlier in the Component Definition dialog box.

```
/*
 * Ball Class. Extends MovieClip.
 *
 * Component Params:
 *   cp_radius          -The ball's size
 *   cp_ballColor       -The ball's color
 *   cp_x               -The ball's initial x position
 *   cp_y               -The ball's initial y position
 *   cp_velocity        -The ball's movement vector: {x:0, y:0}
 *
 * Methods: ...
 *
 * Event Handlers: ...
 */
/*
 * Class Constructor. Parameters are passed by attachMovie( )'s initObj.
 */
org.moock.Ball = function ( ) {
  // Create instance properties.
  this.velocity = this.cp_velocity;

  // Initialize instance.
  this.setPosition(this.cp_x, this.cp_y);
  this.setColor(this.cp_ballColor);
  this.setSize(this.cp_radius);
  this.onEnterFrame = this.move;

  // Remove component parameters from the instance (any property that starts
  // with the prefix cp_ is a component parameters).
  for (var p in this) {
    if (p.indexOf("cp_") != -1) {
      delete this[p];
    }
  }
};
```

Of course, now that we've changed our *Ball* constructor, we must modify our initObject to use the component property names when creating *ballSymbol* instances with *attachMovie( )*. This lets us use one consistent constructor function,

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

whether the component is created manually at authoring time or programmatically at runtime.

Here's the revised *attachMovie( )* approach:

```
var initObj = new Object( );
initObj.cp_radius = 10;
initObj.cp_ballColor = 0xFF0000;
initObj.cp_x = 250;
initObj.cp_y = 250;
initObj.cp_velocity = {x:12, y:4};

// Create a ball.
this.attachMovie("ballSymbol", "redBall", 1, initObj);
```

If all clip instances of a subclass will be generated programmatically using *attachMovie( )* or *duplicateMovieClip( )*, you should design the constructor to accept parameters attached to a single `params` object within the initObject. However, if you are creating components that will be instantiated at authoring time, you should design the constructor to accept individual component parameters starting with the `cp_` prefix (and you should attach parameters of the same name directly to the initObject when instantiating the component programmatically).

14.5.1 Packaging Component Graphics

Many visual components dynamically create their on-screen display at runtime by attaching exported movie clips inside the component instance. When a component that relies on exported assets is transferred from one *.fla* file to another, each asset must be transferred to the target Library. In Flash MX, there is no formal means of packaging a multisymbol component into a single unit. However, the conventional approach is to place an instance of each asset onto a timeline *guide layer* inside the component (see the Flash online Help for a discussion of guide layers). The assets on a guide layer are transferred automatically when the component is dragged between Libraries, but they do not show up in the component at runtime (unless the component attaches them explicitly).

14.5.2 Component Template

The following code presents a generic component template that implements a *MovieClip* subclass. The template does not use a namespace, such as `org.moock`, as we did in our *Ball* example. Nor does the template store all constructor parameters on a single `params` object. These optional techniques can be implemented at your discretion. However, the template expects constructor parameters to be prefixed with `cp_`, and it deletes all component parameters after construction. Though deleting constructor parameters is not mandatory, it is wise coding practice.

```
// CODE PLACED ON FRAME 1 OF someClassSymbol IN THE LIBRARY.

#initclip

/*
 * SomeClass Class. Extends MovieClip
 *   Version: 1.0.0
 *   Desc: Your description goes here
 *
 * Component Params:
 *   cp_param1       -Short description
 *   cp_param2       -Short description
 *
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

```
 * Methods:
 *   someMethod( )    -Short description
 */

/*
 * Class Constructor
 * (Parameters are passed by an initObj or set as component parameters)
 */
_global.SomeClass = function ( ) {
  // Create Instance properties
  this.prop1 = this.cp_param1;
  // Initialize instance
  this.someMethod(this.cp_param2);
  // Remove component params (those with the cp_ prefix) from the instance
  for (var p in this) {
    if (p.indexOf("cp_") != -1) {
      delete this[p];
    }
  }
};

/*
 * Set MovieClip as SomeClass's superclass
 */
SomeClass.prototype = new MovieClip( );

/*
 * Associate the someClassSymbol in the Library with SomeClass
 */
Object.registerClass("someClassSymbol", SomeClass);

/*
 * Instance Methods
 */

/*
 * Method: SomeClass.someMethod( )
 *   Desc: Short description
 *
 * Params:
 *   param     -Short description
 *
 */
SomeClass.prototype.someMethod = function (param) {
  trace("someMethod invoked with parameter of: " + param);
};

#endinitclip
```

The following code shows example usage of the *SomeClass* template. To see a manually created *SomeClass* instance, download the *componentTemplate.fla* file from the Code Depot.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

```
// Create an instance
var initObj = new Object( );
initObj.cp_param1 = 15;
initObj.cp_param2 = "Hello world";
this.attachMovie("someClassSymbol", "someClassInstance", 1, initObj);
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Chapter 14. Movie Clip Subclasses and Components

14.6 MovieClip Sub-Subclasses

Just like a regular class, a *MovieClip* subclass can itself be extended by another subclass. When a parent class is defined outside of its corresponding movie clip symbol (as in the very first *Ball* example in this chapter), its child subclass can be defined by:

- Setting the child's prototype to a new parent instance
- Registering the child class with its symbol via *Object.registerClass ( )*

Remember also to invoke the parent's constructor function from the child constructor. For example, the following code shows how to create a subclass, *Baseball* , of our existing class, *Ball* :

```
// Baseball class constructor
org.moock.Baseball = function ( ) {
  // ... init code ...

  // Invoke Ball class constructor
  super( );
}

// Set Baseball's superclass to Ball
org.moock.Baseball.prototype = new org.moock.Ball( );

// Associate the Baseball class with its own Library symbol
Object.registerClass("baseballSymbol", org.moock.Baseball);
```

However, when both the subclass (*Ball*) and the sub-subclass (*Baseball*) are defined inside a movie clip symbol's `#initclip` block, we must ensure that the subclass is initialized *before* any of the sub-subclasses. To do so, we use the `order` parameter of the `#initclip` pragma, which is a nonnegative integer specifying the execution order of a particular `#initclip` block, relative to all other `#initclip` blocks in the movie. Those `#initclip` blocks with lower `order` numbers are always executed before those with higher ones. All `#initclip` blocks without a specified `order` are executed before those with a specified `order` .

For example, to create a *Baseball* class inside the movie clip *baseballSymbol* , we must change the first line of our *Ball* class code to:

```
#initclip 0
```

Then we can create our *Baseball* class as follows:

```
// Use a higher #initclip order than Ball uses, guaranteeing that Ball
// will be available before this code runs.
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

```
#initclip 1

// =  =  =  =  =  =  =  =  =  =  =  =  =  =  =  =  =  =  =  =  =  =  =  =
// Create namespace in which to store classes
// =  =  =  =  =  =  =  =  =  =  =  =  =  =  =  =  =  =  =  =  =  =  =  =
if (_global.org =  = undefined) {
  _global.org = new Object( );
}
if (_global.org.moock =  = undefined) {
  _global.org.moock = new Object( );
}

// =  =  =  =  =  =  =  =  =  =  =  =  =  =  =  =  =  =  =  =  =  =  =  =
// Create the Baseball class
// =  =  =  =  =  =  =  =  =  =  =  =  =  =  =  =  =  =  =  =  =  =  =  =

/*
 * Baseball Class. Extends Ball.
 *   Version: 1.0.0
 *   Desc: A movie clip subclass for depicting baseballs
 *
 * Constructor Params:
 *   signature       -The ball's signature
 *
 * Methods:
 *   setSignature( )  -Set the signature on the ball
 */

/*
 * Class Constructor. Parameters are passed by attachMovie( )'s initObj.
 */
org.moock.Baseball = function ( ) {
  // Create Instance properties.
  this.sig = null;

  // Initialize instance before calling Ball constructor. (Otherwise,
  // Ball will delete the params object before we get a chance to use it.)
  this.setSignature(this.params.signature);
  // Invoke Ball class constructor.
  super( );

  // No need to delete params object, because the Ball constructor does that.
};
/*
 * Set Ball as Baseball's superclass.
 */
org.moock.Baseball.prototype = new org.moock.Ball( );

/*
 * Associate the Library's baseballSymbol with the Baseball class
 */
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

```
Object.registerClass("baseballSymbol", org.moock.Baseball);
/*
 * Instance Methods
 */

/*
 * Method: Ball.setSignature( )
 *   Desc: Sets the ball's signature
 *
 * Params:
 *   newSig     -The ball's new signature
 */
org.moock.Ball.prototype.setSignature = function (newSig) {
  this.sig = newSig;
};

#endinitclip
```

Note that multiple `#initclip` blocks can share a single *`order`* position. For example, if we create three subclasses of *Ball* , we can set all of their `#initclip` order parameters to 1. All that matters is that the *Ball* 's `#initclip` order is lower than all its subclasses.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Chapter 14. Movie Clip Subclasses and Components

14.7 Summary

Over the course of this chapter, we've studied three related but distinct types of *MovieClip* subclasses:

- A subclass defined outside of its Library symbol
- A subclass embedded in its Library symbol (inside an `#initclip` directive)
- A component, which allows component parameters to be specified at authoring time via a GUI

The type of *MovieClip* subclass you use in your application depends entirely upon your situation. If you must allow for author-time placement of instances, you'll very likely need to create a component. Similarly, if you plan to distribute your subclass to other developers, a component makes for a good transport mechanism. Otherwise, you can skip the component-definition step. In any event, you should normally define your subclass directly inside its associated Library symbol, within an `#initclip` block. While not legally required, storing a subclass in its associated symbol makes the entire package easier to maintain.

Movie clip subclasses and components are two cornerstones of the Flash MX application development process. By creating code libraries as components, we can rapidly assemble applications based on reusable assets. Macromedia bundles a number of useful UI Components with Flash MX. You should familiarize yourself with these components, their methods, and parameters (see Appendix G). The Flash UI Components source code is included with Flash MX, so you can study and extend these components to create your own.

For further component study, see:

Introduction to Components Tutorial

In the Flash MX authoring tool, select Help Tutorials Introduction to Components.

Using Components

In the Flash MX authoring tool, select Help Using Flash Using Components.

Building an Address Book with Components (by Colin Moock)

http://www.macromedia.com/desdev/mx/flash/articles/addressbook.html

Creating Forms with Components in Macromedia Flash MX

http://www.macromedia.com/support/flash/applications/creating_forms/

Creating Components in Macromedia Flash MX

http://www.macromedia.com/support/flash/applications/creating_comps/

Creating a Flash MX Component

http://www.flashcomponents.net/tutorials/triangle/triangle.html

Extending Components in Macromedia Flash MX

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

http://www.macromedia.com/desdev/mx/flash/articles/fmx_components.html

Customizing the ListBox and ComboBox Components

http://www.macromedia.com/desdev/mx/flash/articles/extending_components.html

Macromedia's Flash Component Exchange

http://www.macromedia.com/exchange/flash

Third-Party Component Exchange

http://www.flashcomponents.net/

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

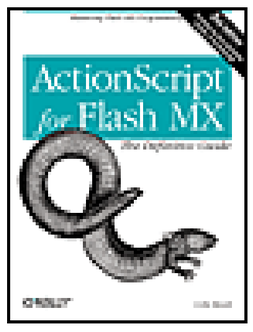
Chapter 14. Movie Clip Subclasses and Components

14.8 Onward!

Components and movie clip subclasses were our last major topic of study in ActionScript's language fundamentals. Before we move on to the ActionScript Language Reference, we'll study *lexical structure* (the finicky details of ActionScript syntax), and we'll put things together in a final application tutorial, Chapter 17. See also Chapter 16 for more information on the component definition process.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir



ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Part I: ActionScript Fundamentals

Chapter 15. Lexical Structure

The *lexical structure* of a language is the set of rules that govern its syntactic composition. We must follow these rules when writing the source code of our scripts.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Chapter 15. Lexical Structure

15.1 Whitespace

The tab, space, and carriage return (i.e., line break) characters are used in ActionScript just as they are used in English: to separate words from each other so that they don't all runtogetherlikethis. In programmer-speak, these characters are known as *whitespace* and are used in source code to separate *tokens* (the keywords, identifiers, and expressions roughly akin to words, phrases, and sentences in English). Here is an example of incorrect and correct whitespace usage:

```
varx;   // Oops! No whitespace between the var keyword and the variable x.
var x;  // That's better...because of the whitespace, the interpreter
        // can now understand our code.
```

Whitespace is optional when there is some other *delimiter* (separator) that tells ActionScript where one token ends and another begins. The following code is quite legitimate, because the operators =, +, and / separate `x`, 10, 5, and `y` from one another:

```
x=10+5/y;         // Crowded, but legitimate
x = 10 + 5 / y;  // Easier to read, but the same as above
```

Similarly, whitespace is optional when there are other characters—such as square brackets, parentheses, curly braces, commas, and greater-than or less-than signs—to act as delimiters. These are all perfectly legal, if somewhat claustrophobic:

```
for(var i=0;i<10;i++){trace(i);}
if(x=  =7){y=[1,2,3,4,5,6,7,8,9,10];}
myMeth=function(arg1,arg2,arg3){trace(arg1+arg2+arg3);};
```

Extra whitespace is simply a matter of style, because it is ignored by the ActionScript interpreter. That said, there are conventions you should follow to make your code more readable. For example, the following is another legitimate way to rewrite the earlier assignment expression, but it is obviously harder to read:

```
x =
  10
  + 5
  / y;
```

Notice that the statement is terminated by the *semicolon*, not the line breaks. In nearly all cases, line breaks are inconsequential and do not act as statement terminators. Hence, we often use one or more line breaks (in concert with spaces or tabs) to make complex statements easier to read:

```
myNestedArray = [[x, y, z],
                 [1, 2, 3],
                 ["joshua davis", "yugo nakamura", "james patterson"]];
// Much nicer than:
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

```
myNestedArray = [[x, y, z], [1, 2, 3],["joshua davis","yugo nakamura",
"james patterson"]];
// And also nicer than:
myNestedArray = [[x, y, z],
[1, 2, 3],
["joshua davis","yugo nakamura","james patterson"]];
```

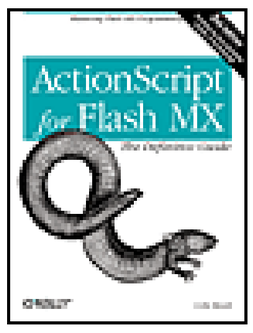
You don't need to do anything special to break a statement onto more than one line. Simply add a carriage return and keep typing. For good advice on making your code more readable, see *Code Complete*, by Steve McConnell (Microsoft Press). However, there are times when line breaks may be misinterpreted as statement terminators, as we'll see in the next section.

Note that whitespace *within* the quotes that delimit a string is relevant, whereas whitespace *outside* the quotes is ignored. Compare these two examples:

```
x = 5;
trace("The value of x is"  +   x);   // Displays: "The value of x is5"
trace("The value of x is "+x);       // Displays: "The value of x is 5"
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir



ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Chapter 15. Lexical Structure

15.2 Statement Terminators (Semicolons)

As we saw in Chapter 6, the semicolon terminates an ActionScript statement. Though by convention you should always end your statements with semicolons, they are not strictly required in ActionScript. If the semicolon is omitted, the interpreter attempts to infer the end of a statement. For example:

```
// These are preferred
var x = 4;
var y = 5;
// But these are also legal
var x = 4
var y = 5
```

The ActionScript interpreter assumes that the line breaks in this code are intended as statement terminators. (Compilers in stricter languages such as C would complain.) However, omitting semicolons from statements in code is somewhat like omitting periods from sentences in normal writing—the reader will probably understand most of your sentences, but there will always be cases that lead to confusion. Not to mention it's more taxing to read. For example, consider what happens when we omit a semicolon after the *return* statement:

```
function addOne (value) {
  return
  value + 1
}
```

ActionScript will assume that we meant to write this:

```
function addOne (value) {
  return;
  value + 1;
}
```

Instead of returning `value + 1`, the function will always return `undefined`, because the keyword `return` alone is a legal statement. Even if "`return`" appears alone on a line, and even if we add a semicolon after `value + 1`, the *return* statement is still treated as a complete statement.

To avoid this type of ambiguity, it's good practice to include semicolons. Furthermore, in the specific case of the *return* statement, don't separate the keyword `return` from its expression with a line break, as that alters the statement's meaning. Therefore, the preceding example should be written as:

```
function addOne (value) {
  return value + 1;
}
```

Similarly, a postfix `++` or `--` operator should always appear on the same line as its operand:

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

```
// Right
x++
// Wrong
y
++
```

Note that semicolons terminate individual statements but are not required where statement block delimiters (curly braces) occur. For example:

```
for (var i=0;i<10;i++) {  // No semicolon here
  trace(i);               // Semicolon here
}                         // No semicolon here

if (x =  = 10) {            // No semicolon here
  trace("x is ten");      // Semicolon here
} else {                  // No semicolon here
  trace("x is not ten");  // Semicolon here
}                         // No semicolon here

on (release) {            // No semicolon here
  trace("Click");         // Semicolon here
}                         // No semicolon here
```

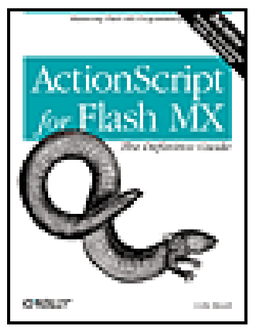
However, the semicolon is mandatory following a function literal:

```
function (param1, param2, ... paramn) { statements };
```

The special `#include` directive, and the `#initclip` and `#endinitclip` pragmas should not be followed by a semicolon and will cause an error if used with one. For correct usage, see the *Language Reference*.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir



ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Chapter 15. Lexical Structure

15.3 Comments

So-called *comments* are text that is ignored by the interpreter but entered by the programmer specifically for the benefit of humans reading the code. Comments should be used liberally to explain what the code does, provide version control information, or describe other relevant information, such as how a data structure is being used or why certain programming choices were made. Comments should describe your code at the conceptual level, not merely mirror the syntax of the code itself. For example, the following comment is useless:

```
// Set x equal to 5
var x = 5;
```

whereas the following comment tells us *why* we're setting `x` to 5, which helps us follow the flow of the code:

```
// Initialize our player's x position
var x = 5;
```

Note that you can go a long way toward making your code "self-commenting" by using descriptive variable names and handler names. Consider which of the following is clearer:

```
x = y / z;                          // This is very cryptic
average = sum / numberOfItems;      // This hardly needs explanation
```

ActionScript supports both one-line comments and extended, multiline comments. One-liners, as we've seen throughout this book, begin with two forward slashes (`//`):

```
// Ah, human contact...this will never be read by the interpreter
```

One-line comments are automatically terminated by the line break at the end of the line. You must repeat the `//` characters to add more comment text on the following line:

```
// Here is the start of a comment...
// ...and here's some more of it.
```

One-line comments can also follow real code on the same line:

```
var x;    // This is a valid comment on the same line as code
```

ActionScript also supports multiline comments, used to accommodate larger blocks of code commentary, that are *not* terminated by line breaks. Multiline comments start with the two-character sequence `/*` and continue until the terminating two-character sequence `*/` is encountered, as in:

```
/* -----BEGIN VERSION CONTROL INFO----
  Name: MyApplication, Version 1.3.1
  Author: Killa Programma
  Last Modified: August 07, 2002
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

```
   ------END VERSION CONTROL INFO-----
*/
```

Multiline comments are available only in Expert Mode in the Actions panel or in external *.as* source files included via the `#include` directive. Multiline comments are converted to single-line comments when you switch from Expert Mode to Normal Mode. In Flash MX, switching back to Expert Mode restores multiline comments. Note that nested comments are legal:

```
/* This is an example of
/* a nested // comment */

/* This another
/* example of nested comment */
The comment doesn't end until the second terminator is encountered here */
```

The following is avoided by convention, but it is valid:

```
/* This comment is followed by real code */ var x = 5;
```

When making major (or even minor) revisions to some code, it is common practice to include your initials and the date of the change in a comment, such as this:

```
// Fixed error in financial calculation. BAE 12-04-02. V1.01
```

Comments let us temporarily disable code (or permanently disable old code) instead of deleting it. Disabling code by making it into a comment is known as *commenting out* the code. It is easy to reenable the code by removing the comment delimiters:

```
/* Disable this block of code until I change my mind
duplicateMovieClip("character", "newCharacter", 1);
newCharacter._rotation = 90;
*/
```

You can disable a single line of code by simply preceding it with two slashes:

```
// newCharacter._rotation = 90;
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Chapter 15. Lexical Structure

15.4 Reserved Words

Some *reserved words* are used by the ActionScript interpreter to denote specific built-in language features, such as statements operators. They are reserved for use by the interpreter, and we must avoid using them as identifiers in our code. Using a reser word for any purpose other than its reserved internal purpose causes an error in most cases. ActionScript's reserved words are listed in Table 15-1.

Table 15-1. ActionScript's reserved words

add[1]	delete	ge[1]	le[1]	onClipEvent	this
and[1]	do	gt[1]	lt[1]	or[1]	typeof
break	else	if	ne[1]	return	var
case	eq[1]	ifFrameLoaded[1]	new	super	void
continue	for	in	not[1]	switch	while
default	function	instanceof	on	tellTarget[1]	with

[1] Flash 4 reserved words that have been deprecated as of Flash 5.

You should also try to avoid using the keywords listed in Table 15-2. They are not reserved words for ActionScript in Flash M but they may become a part of the language in the future, because they are slated for potential use by ECMA-262.

Table 15-2. Potential reserved words in the future

abstract	const	final	int	protected	transient
boolean	debugger	finally	interface	public	try
byte	double	float	long	short	volatile
catch	enum	goto	native	static	
char	export	implements	package	synchronized	
class	extends	import	private	throws	

In addition to the formally defined reserved keywords, you should also avoid using the names of built-in properties, methods, and objects as identifiers in your code. Doing so overrides the default behavior of the property, method, or object in question.

The following is a bad idea because the `Date` variable overrides the built-in constructor of the same name:

```
Date = new Object();   // Oops! We just disabled the Date( ) constructor
```

downloaded from: lib.ommolketab.ir

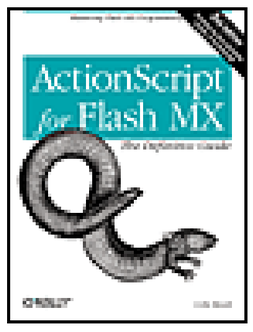
downloaded from: lib.ommolketab.ir

Now we can no longer create *Date* objects:

```
var now = new Date( );  // Sets now to undefined
trace(now);             // Displays the empty string, not the current time and
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir



ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Chapter 15. Lexical Structure

15.5 Identifiers

All variables, functions, and object properties are named with identifiers. ActionScript identifiers must be composed according to the following rules:

- Identifiers must start with a letter, underscore, or dollar sign (not a number).
- Identifiers cannot be identical to reserved words.
- For safety, identifiers should include only letters (*A-Z* or *a-z*), numbers, underscores, and dollar signs. Be especially careful never to use spaces, periods, backslashes, or other punctuation in identifiers. However, most Unicode characters are, in fact, permitted in identifiers, according to the rules outlined in section 3.16 of the Unicode standard. (See http://www.unicode.org/unicode/standard/standard.html).
- A character in an identifier cannot be specified with a Unicode escape sequence (\uxxxx). (This is a departure from ECMA-262, v3, which allows escape sequences in identifiers.)
- See the next section on case-sensitivity for more caveats about identifiers.

Though not strictly required, it's also good practice to follow these rules when composing movie clip instance names, symbols, symbol export identifiers, frame labels, and layer names. For information on using standardized suffixes in identifier names (such as `_mc` to indicate a movie clip) see Table 2-1 in Chapter 2.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Chapter 15. Lexical Structure

15.6 Case Sensitivity

When a language is fully case sensitive, every token in the language—including all identifiers and keywords—must be entered with the correct capitalization. For example, in case-sensitive languages, the statement:

```
If (x =  = 5) {
  x = 10;
}
```

would cause an error, because the `if` keyword is improperly capitalized as `If`. Furthermore, in case-sensitive languages, the following two statements would declare two separate variables (one named `firstName` and a second one named `firstname`):

```
var firstName = "doug";
var firstname = "terry";
```

The ECMA-262 specification, upon which ActionScript is based, demands complete case sensitivity. ActionScript, however, diverges from the standard in the area of case sensitivity in order to maintain backward compatibility with Flash 4 movies. In ActionScript, the keywords in Table 15-1 are case sensitive, but identifiers, frame labels, and symbol export IDs are not. In the following example, the `onClipEvent` keyword is incorrectly capitalized as `onclipevent` and causes an error:

```
// ERROR! Should be onClipEvent (enterFrame) {
onclipevent (enterFrame) {
  this._x++;
}
```

But unlike keywords, identifiers are not case sensitive in ActionScript, so the following statements assign a value to the *same* variable:

```
var firstName = "margaret";
var firstname = "michael";
trace(firstName);   // Yields "michael"
trace(firstname);   // Also yields "michael" (the variables are the same)
```

Even internal identifiers, such as property names and function names, are not case sensitive in ActionScript. The following line causes an error in JavaScript but successfully creates an array in ActionScript:

```
myList = new array( );   // Should preferably be new Array( );
```

This can present problems when porting JavaScript code to ActionScript or for JavaScript programmers learning ActionScript. Case sensitivity is often used in JavaScript to apply the same name to different purposes. For example, the statement:

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

```
date = new Date( );   // Works in JavaScript but not ActionScript
```

will have destructive effects in ActionScript where the identifier `date` is not distinguished from the object class *Date*. In ActionScript, the preceding code disables the built-in *Date* class. We must therefore ensure that our identifiers are distinguished by more than just case, both from one another and from any predefined identifiers, such as *Date* or *Array*. Here's how we rewrite the previous code in ActionScript:

```
theDate = new Date( );  // Use this approach in ActionScript
```

Frame labels and symbol export IDs are also not case-sensitive. For example, the following *gotoAndPlay( )* function will work even if the destination label is `INTRO` instead of `Intro`:

```
gotoAndPlay("Intro");
```

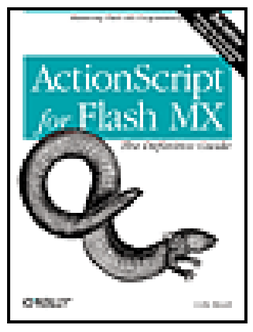
Similarly, a symbol exported as "SHIP" can be attached as follows:

```
this.attachMovie("ship", "enemyShip", 1);
```

However, these case mismatches are not recommended practice. The key in all situations is to be consistent—even when consistency is not strictly required by the language. Consistently capitalizing variables, functions, instances, and other items makes our code easier to read and prevents future changes in ActionScript's case rules from breaking our hard work.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir



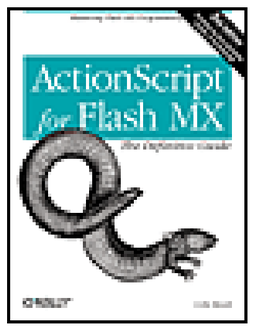
Chapter 15. Lexical Structure

15.7 Onward!

As far as ActionScript language fundamentals go, we've reached the end of the road. You're now fully equipped to both read and write ActionScript. Chapter 16 covers the ActionScript authoring environment.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir



ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Part I: ActionScript Fundamentals

Chapter 16. ActionScript Authoring Environment

This chapter examines the practical details of authoring ActionScript code. We'll cover the following topics:

- Using the Actions panel to place code on buttons, movie clips, and frames
- Loading code from an external file
- Creating components (reusable code modules)

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Chapter 16. ActionScript Authoring Environment

16.1 The Actions Panel

The *Actions panel* is Flash's ActionScript-editing environment. Every script in a movie is created in the Actions panel, which is accessed via Window → Actions (F9).

The Actions panel can operate in Expert Mode or Normal Mode. When in Expert Mode, the panel is divided into two sections: the Toolbox pane (on the left) and the Script pane (on the right), as shown in Figure 16-1.

Figure 16-1. The Actions panel

The Script pane houses code attached to the currently selected frame, button, or movie clip. The Toolbox acts as both a quick reference guide and a means of adding code to the Script pane. Double-clicking any item in the Toolbox adds that item to the Script pane. Items can also be dragged from the Toolbox into the Script pane.

The Actions panel's title bar indicates whether our current code lives on a frame ("Actions - Frame"), a button ("Actions - Button"), or a movie clip ("Actions - Movie Clip"). When code cannot be attached to the current selection, the Actions panel title is simply "Actions", and the Script pane is disabled.

The Actions panel's pop-up Options menu button provides access to various tools and settings, such as printing and enabling line numbers. The script navigation bar indicates the exact location of the current script and can be used to navigate quickly between scripts on the current timeline.

The organization of the items in the Toolbox differs somewhat from the categories used in this book to describe the

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

features of ActionScript. Most notably, in the Toolbox, *statements* are not separated into their own folder, and classes and objects are grouped together under the *Objects* folder. In this book, we distinguish between statements, classes, and objects in keeping with more traditional programming terminology. For an alphabetical list, use the index at the bottom of the Toolbox or, of course, this book's ActionScript Language Reference.

16.1.1 Editing Modes

The Actions panel has two different modes of operation, Normal Mode and Expert Mode, which govern how we add code to the Script pane.

16.1.1.1 Normal Mode

In Normal Mode, the Toolbox is used to build new statements, and the Script pane is used as a viewer for those statements. To add a new statement to the Script pane in Normal Mode, we double-click the desired Action or drag an Action from the Toolbox to the Script pane. To add a statement of arbitrary code, we select the *evaluate* Action (under Actions → Miscellaneous Actions) and enter the statement into the Expression field of the Parameters pane at the top of the Actions panel (the Parameters pane is not shown in Figure 16-1).

After adding a statement to the Script pane, we can customize it via the Parameters pane at the top of the Actions panel. The layout and contents of the Parameters pane change to suit the statement selected in the Script pane. Notice that when the Actions panel is operating in Normal Mode, code *cannot* be entered directly into the Script pane. In Normal Mode, the Script pane is not a text-editing environment, but rather a read-only list of statements; alterations to statements in the Script pane are performed through the Parameters pane.

Normal Mode is comforting for new programmers but constricting for most nontrivial programming projects. In this book, therefore, we use only the Expert Mode to author code. However, with the exception of multiline comments (which can be used only in Expert Mode), any code produced in one mode can also be produced in the other.

16.1.1.2 Expert Mode

In Expert Mode, the Script pane functions as a traditional text-editing window. When authoring or changing code in Expert Mode, we enter text directly into the Script pane. However, we can still double-click items in the Toolbox to add them to the Script pane. The Parameters pane is not used in Expert Mode; instead, parameters are typed directly into the Script pane.

16.1.1.3 Setting the editing mode

In Flash 5, the editing mode of the Actions panel is set on a per-frame and per-object basis. That is, Flash remembers the editing mode chosen for every object and frame in a movie. If we select Normal Mode for frame 2 and Expert Mode for frame 3, the Actions panel automatically switches to Normal Mode when we edit the code on frame 2 and Expert Mode when we edit the code on frame 3.

In Flash MX, the editing mode of the Actions panel is a permanent toggle, accessed via the pop-up Options menu shown in Figure 1-2 in Chapter 1.

In Flash 5, the editing mode can be set globally under Edit Preferences General Actions Panel Mode.

In Flash 5, switching from Expert Mode to Normal Mode destroys all source-code formatting. The code is reformatted according to Flash's standard formatting rules—extra whitespace is lost, and statement block indentation is adjusted to match the style shown throughout this book. In Flash MX, source-code formatting is preserved between

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

mode switches.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Chapter 16. ActionScript Authoring Environment

16.2 Adding Scripts to Frames

Flash documents are fundamentally structured around the concept of animation. Every Flash document comprises a linear sequence of *frames*, or slices of visual and audio content. A great deal of ActionScript code is tied to this frame-based structure. By placing a code block on a frame, we stipulate the timing of that block's execution relative to the playback of the movie. Whenever we add code to a frame, we must always consider not only what we want the code to do, but also *when* we want the code to execute.

For example, we may want to write some code that places a new movie clip on stage when the 20 th frame of a movie is displayed:

```
_root.attachMovie("myClip", "clip1", 0);
```

In order for this code to execute when the 20 th frame is displayed, we must attach it to a keyframe at frame 20 of the movie.

> ActionScript code attached to a frame must reside in a keyframe. Use Insert Keyframe or Insert Blank Keyframe to create a keyframe to which your code can be attached.

To attach code to a keyframe, we select the keyframe in the timeline, open the Actions panel, and then add the desired code to the Script pane. During playback, code on a keyframe is executed before the content of the frame is displayed.

Code on keyframes is used to perform tasks synchronized to the movie's playback and to establish program elements such as variables, functions, and objects for use throughout a movie clip or movie. A timeline loop is a simple example of a synchronized task. Suppose we attach the following code to frame 15 of a movie:

```
gotoAndPlay(10);
```

When the playhead of the movie reaches frame 15, the code executes, and the movie begins playing back at frame 10 again. When frame 15 is reached for the second time, the code is executed again, and the movie again starts playing at frame 10. This causes the movie to cycle endlessly between frames 10, 11, 12, 13, 14, and 15.

However, code on keyframes does not always control or synchronize with a movie's playhead. We can also use keyframes to represent application states (e.g., as a stopping point in a movie, such as for each quiz question in Chapter 1), or simply as storage devices for functions, variables, and objects. For example, we can attach the function *moveClipTo( )* to frame 1 of a movie clip and invoke *moveClipTo( )* from a button later in the movie:

```
function moveClipTo (x, y) {
  _x = x;
  _y = y;
}
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

One caution: because the execution of code on frames is dictated by the playback of a movie, we must always ensure that variables, functions, and other program elements are available before they are accessed. For example, we can't invoke a function at frame 3 if that function isn't defined until frame 10. Code used globally throughout a movie should therefore be placed on the first frame of the main timeline or on the first frame after the movie's preloader. See the `_global` object in the *Language Reference* for more information on global code placement. See also Chapter 2 and Chapter 9 for a discussion of variable and function scope.

We must also ensure that no code attempts to access portions of a movie that have not yet loaded. To check whether a specific portion of a movie has loaded, use either the movie clip `_framesloaded` property or the *getBytesLoaded( )* method. For sample code, see the entry for `Movieclip._framesloaded` in the ActionScript Language Reference.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Chapter 16. ActionScript Authoring Environment

16.3 Adding Scripts to Buttons

In Flash 5, button code is attached directly to button objects on stage. For example, in Flash 5, if we want the clicking of a button to advance the playhead of a movie to a frame labeled `section1`, we attach the following code to that button:

```
on (release) {
  gotoAndStop("section1");
}
```

In most cases, Flash MX's button callback-function-based event handlers should be used on a movie clip timeline in place of the older style *on(event)* syntax. However, in some trivial situations it is more convenient to add code directly to a button.

To add code to a button, select the button on the Stage, and add the code to the Script pane of the Actions panel. Code on buttons must always be placed in an event handler that identifies the circumstances under which the code should execute. For example, the event that triggers most button actions is *release*. We might also use the *rollOver* event, causing code to execute when the mouse moves over the button, not when the button is pressed:

```
on (rollOver) {
  this.gotoAndStop("section1");
}
```

For a complete description of button event handlers, see Chapter 10.

Though it is legal to place thousands of lines of code on a button, it's always a bad idea to overload a button with code. Whenever possible, generalize and package the behavior of button code into functions attached to the button's parent timeline. For example, we could add the following code to a button:

```
on (release) {
  this.title._xscale = 20;
  this.title._yscale = 20;
  this.title._alpha  = 50;
  this.title.gotoAndPlay("fadeout");
}
```

But we're better off placing that code in a function and calling the function from the button:

```
// CODE ON FRAME 1 OF THE MOVIE. ADDS A fadeOut( ) METHOD TO ALL CLIPS.
MovieClip.prototype.fadeOut = function (scale, transparency) {
  this._xscale = scale;
  this._yscale = scale;
  this._alpha = transparency;
  this.gotoAndPlay("fadeout");
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

```
}
// CODE ON BUTTON
on (release) {
  this.title.fadeOut(20, 50);
}
```

Or, better yet, in Flash MX we should add an event handler to the *Button* instance as follows:

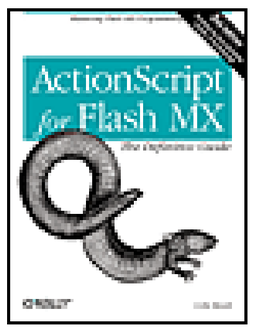
```
theButton.onRelease = function ( ) {
  this._parent.title.fadeOut(20, 50);
}
```

This approach keeps all of our code centralized and easy to maintain and allows us to apply the button's behavior to many buttons quickly and with minimal effort.

Button code must be attached to a button object on stage and must include an event handler. Button code without an event handler will cause an error, and it's not possible to attach code to the internal Up, Over, Down, and Hit frames of a button symbol.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir



ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Chapter 16. ActionScript Authoring Environment

16.4 Adding Scripts to Movie Clips

We've already seen that we can attach code to the frames of a movie clip's timeline. It is also possible to attach code to a movie clip instance itself. To do so, we select the instance on stage and then add our code to the Script pane of the Actions panel. As with buttons, all code attached to movie clip objects must be contained within event handlers. Event handlers tell the interpreter when to execute the movie clip code. For example, the following code sets the variable `x` to 10 when a movie clip loads:

```
onClipEvent (load) {
  var x = 10;
}
```

Movie clip event handlers can react to mouse and keyboard activity, data loading, frames rendering, and the birth and death of movie clips. For complete coverage of movie clips and event handlers, see *MovieClip* in the *Language Reference*.

> With the introduction of the callback-function event model in Flash MX, attaching code directly to movie clip instances is rarely necessary. Event handlers should be applied to movie clip instances either from a central code repository on the main timeline or, if the handler applies to all instances of the symbol, from within the movie clip symbol itself. Event handlers can also be specified for an entire movie clip symbol with a *MovieClip* subclass (see Chapter 14). For very specific exceptions, consult the *MovieClip* event handler entries in the ActionScript Language Reference.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Chapter 16. ActionScript Authoring Environment

16.5 Where's All the Code?

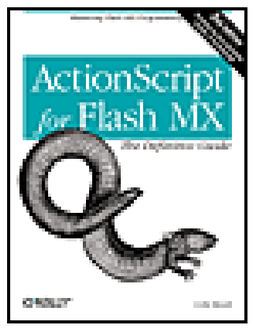
Even for experienced Flash users, locating the code in a movie can sometimes be challenging. Because code can be attached to any frame in any timeline, any button, or any movie clip on stage, it is often lost in a sea of content. Obviously, a highly organized structure and thorough code documentation can save dozens or hundreds of hours over the life span of a project. But if you're faced with a movie that mysteriously seems to be missing important code, open the Actions panel and try these techniques for finding it:

- To search the current scene for ActionScript code, use the Movie Explorer window's search option as described in the paragraph that follows this list.
- Click on any frame that has a little circle icon on it in the timeline. A circle indicates the presence of ActionScript code.
- Look for, and select, any white circle with a black outline on stage. These circles indicate empty movie clips, which often contain nothing but code. If there's no code on the empty clip itself, double-click the clip to edit it and investigate its frames.
- Select each button in the movie, one at a time. Some programmers tend not to centralize code and instead place long, important scripts directly on buttons.
- Check the timeline for hidden or masked layers. A layer with a red X icon next to it is hidden during authoring but may contain clips and buttons with code that appear during movie playback. Similarly, masked layers may contain obscured objects that bear code. Unlock masked layers to reveal their contents.
- Unlock all layers. Empty movie clips (the little circles with black outlines) are hidden when the layer they're on is locked.
- Use Ctrl+F (Windows) or Command+F (Macintosh) to open the Find dialog box. It is useful for searching large scripts for the specified text. Use the Movie Explorer, as described next, to search across all scripts.

Sometimes even these techniques can fail us. If someone's determined to hide some code, there are lots of places in Flash to do so. For example, an empty clip can be placed far beyond the limits of the Stage, making it nearly impossible to find. Fortunately, when all seems lost, our code isn't—we can always use the Movie Explorer to hunt down any script in a movie. Select Window Movie Explorer for a bird's-eye view of all the assets in the current scene, including any scripts attached to a frame, button, or movie clip. Make sure the Show Action Scripts icon is depressed under the Show menu at the top of the Explorer panel, so that the search includes scripts. You can use the same window to search for text fields containing a certain word. There is no easy way to search the entire movie; you'll need to search each scene separately. Scripts are marked with a blue arrow Actions icon similar to the icons in the Actions panel Toolbox. To filter the display to show only scripts, select the Show Action Scripts icon under the Show menu, and deselect all other icons.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir



ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Chapter 16. ActionScript Authoring Environment

16.6 Productivity

Here are some tips to streamline ActionScript source code authoring:

- Keep all timeline scripts on an independent layer called *scripts*. Do not place any content on the *scripts* layer; use it exclusively for code. Store this layer at the top of your timeline panel so that its code executes after all other layers have loaded (or at the bottom if you set your Load Order to Top Down). Keeping the *scripts* layer in the same place makes your code easy to find. (We saw in the previous section how valuable that can be!)

- Keep all frame labels on an independent layer called *labels*. Do not place any content on the *labels* layer; use it exclusively for frame labels.

- Take a look under the pop-up Options menu button in the top-right corner of the Actions panel. You'll find handy tools such as search and replace, source code printing, and Script pane font control.

- When working with code libraries used on multiple projects, save code in external files. See Section 16.7 for details.

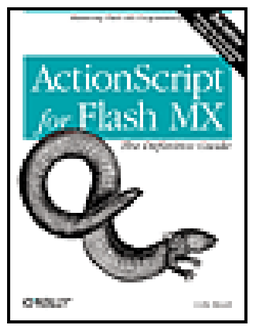
- To save typing, use the shortcut keys provided for Actions; for example, use Esc+G+P for *gotoAndPlay( )*. Shortcut sequences are listed under the plus (+) button in the Actions panel.

- Notice that the Toolbox is resizable. To save space while coding, you can hide the Toolbox, partially or completely, by dragging the border between the Toolbox and the Script pane.

- For a brief reference description of any item in the Toolbox, select the item, and then click the Reference (book) icon in the top right corner of the Actions panel.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir



ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Chapter 16. ActionScript Authoring Environment

16.7 Externalizing ActionScript Code

ActionScript code can be saved in external text files (which use the *.as* extension, by convention) or *.swf* files and imported into a Flash document. By maintaining code in external files, we facilitate the use of standard code libraries across many projects. We can import external code into Flash using Import From File, the `#include` directive, or a shared library.

16.7.1 Import From File (Author-Time Import)

While editing a *.fla* file, we can bring code into the Action panel's Script pane using Import From File, found under the pop-up Options menu button in the top-right corner of the Actions panel (see Figure 16-1). Import From File is a one-time operation that copies the contents of the external file into the Actions panel, *replacing* any script currently there. Code imported via Import From File is not persistently linked to the *.fla* file in any way. To append or insert script text instead of replacing it, you must manually copy and paste the text from an external text-editing application.

16.7.2 #include (Compile-Time Import)

When we export (*compile*) a *.swf* file from a *.fla* file, we can incorporate code from an external text file using the `#include` directive. For information on using `#include`, see the *Language Reference.*

16.7.3 Shared Library (Runtime Import)

To import code from an external source while a movie is actually playing, we must create a shared library *.swf* file with a movie clip containing the code to import. Runtime import offers the most flexible approach to sharing code across movies, because it does not require recompilation of the movies that import the shared code; when the shared library *.swf* file is updated, the movies that link to it reflect the update automatically.

The following procedures describe how to share a simple test function from a movie called *codeLibrary.swf* to a movie called *myMovie.swf*. We'll create the *codeLibrary.swf* movie first:

1. Create a new Flash document.
2. Create a new movie clip symbol named *sharedFunctions*.
3. On frame 1 of the *sharedFunctions* clip, add the following code:

```
function test () {
  trace("The shared function, test, was called.");
}
```

4. Select the *sharedFunctions* clip in the Library.
5. From the Library panel, select Linkage from the pop-up Options menu.
6. Select Export for Runtime Sharing.
7. In the Identifier box, type **`sharedFunctions`**.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

8. In the URL box, type `codeLibrary.swf`.
9. Click OK.
10. Save the document as *codeLibrary.fla*.
11. Use File → Export Movie to create *codeLibrary.swf* from *codeLibrary.fla*.
12. Close the *codeLibrary.swf* and *codeLibrary.fla* files.

Now, we'll create the *myMovie.swf* file, which will execute the code imported from *codeLibrary.swf*:

1. Create a new Flash document.
2. Save the new document as *myMovie.fla* in the same folder as *codeLibrary.fla*.
3. Rename *Layer 1* to *sharedCode*.
4. Select File → Open As Library, and choose *codeLibrary.fla*. The *codeLibrary.fla* Library appears.
5. From the *codeLibrary.fla* Library, drag an instance of the *sharedFunctions* clip onto the Stage at frame 1 of *myMovie.fla*.
6. Name the instance `sharedFunctions`.
7. On the main timeline of *myMovie.fla*, create a new layer called *scripts*.
8. Add a keyframe to the *scripts* layer at frame 2.
9. Add a new frame to the *sharedCode* layer.
10. On frame 2 of the *scripts* layer, attach the following code:

```
stop();
sharedFunctions.test();
```

11. Export *myMovie.swf*.
12. The following text appears in the Output window: "The shared function, test, was called."

Note that a shared library is a linked asset, much like a *.gif* image is linked to a *.html* file. Therefore, shared library *.swf* files must always be uploaded correctly with the files that use them. To change the location of a link from an imported symbol to a shared library, follow these steps:

1. Select the symbol in the Library.
2. From the Library panel's pop-up Options menu, choose Linkage.
3. Under URL, set the new location.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Chapter 16. ActionScript Authoring Environment

16.8 Defining Components

A component is a movie clip that allows some of its properties to be assigned through a special graphical user interface in the Flash authoring tool. Components allow nonprogrammers to customize programmatically-controlled movie clips. Components separate the behavior-determining variables from the code of a clip, which lets people treat them as "black boxes"—their operation can remain mysterious as long as their inputs, outputs, and behavior are known.

Components are Flash MX's new and improved version of Flash 5's Smart Clips.

In Chapter 14, we saw how a component can be defined for a *MovieClip* subclass. Our intention in Chapter 14 was to provide an easy-to-use GUI for constructing instances of a subclass. We'll now focus more directly on the process of defining component parameters for any movie clip, not just *MovieClip* subclasses.

Normally, variable initialization occurs in the source code of a movie clip. For example, here we set up the variables used to control a fireworks effect:

```
// User-defined variables
this.numSparks = 10;            // Number of spark clips in the explosion
this.randomDispersion = true;   // Explosion style (true for random,
                                // false for uniform)
this.duration = 1300;           // Length of explosion, in milliseconds
```

Modifying source code of this sort can be intimidating for nonprogrammers. But if we build our code into a component, nonprogrammers can configure the fireworks effect through a familiar application-style interface in the Property inspector. Figure 16-2 shows a component interface equivalent to our variable-initialization code.

Figure 16-2. A sample component-configuration interface

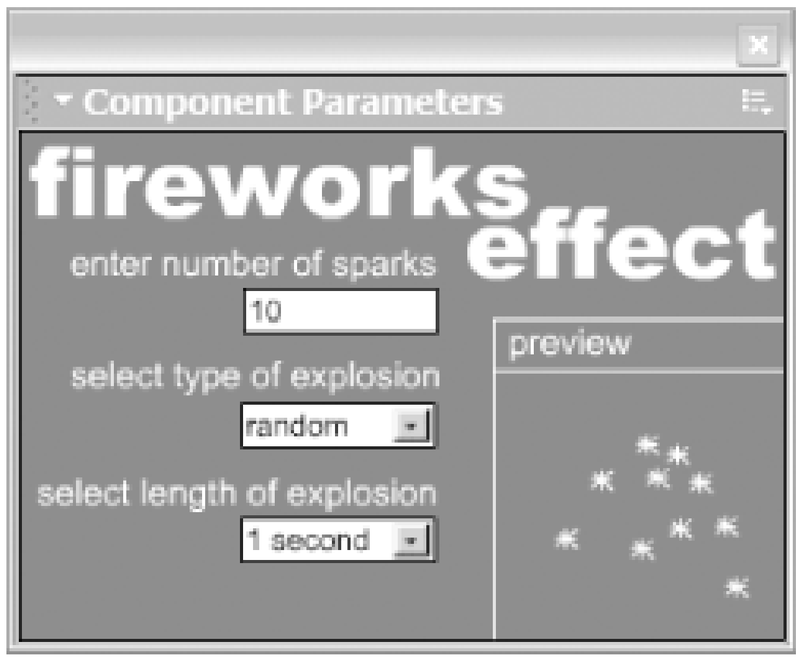
In a component interface, each variable appears with its name and value clearly distinguished in separate rows and columns. Variable names cannot be edited, so there's no chance of inadvertent typographical errors breaking the system. Each variable also has its own verbose name explaining more plainly what it does. Finally, variables with a limited set of legal values (such as `randomDispersion`) can be assigned via drop-down menus.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

For nonprogrammers, the interface shown in Figure 16-2 is certainly more approachable than source code. But components can be made even more user friendly. We can replace the default component interface with our own custom interface, such as the one shown in Figure 16-3. Notice how the custom interface hides our component's variables entirely, allowing the nonprogrammer to tailor each instance of the fireworks effect with text fields and pull-down menus. The interface even provides a preview of the effect in action!

Figure 16-3. A customized component-configuration interface



Let's see how all this works.

16.8.1 Building a Component with a Standard Interface

As we've just seen, components have either the standard system interface or a customized interface. We'll show how to build the standard kind first.

The first step in building any component is creating a regular movie clip that is controlled by the value of one or more variables. In the following code, for example, the variables `xPos` and `yPos` determine the location of a clip on stage:

```
this._x = this.xPos;
this._y = this.yPos;
```

When we build a movie clip as a component for ourselves or someone else to use, we expect certain designated variables to be set via the Property inspector when the clip is placed on stage. These variables are known as *component parameters*. Once we've created a clip with behavior dictated by one or more component parameters, we must give the clip a component interface through which these parameters can be set.

16.8.1.1 Adding a standard interface to a component

To add a default component interface to a movie clip, follow these steps:

1. Select the clip in the Library.
2. Choose Component Definition from the Library's pop-up Options menu. (The Component Definition dialog box appears.)
3. Above the Parameters pane, click the plus (+) button to add a component parameter.
4. Repeat step 3 for each parameter in the component.
5. Configure the component parameters, as described in the following section.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

16.8.1.2 Configuring standard component parameters

After we add a component parameter to a component, we must assign the parameter an on-screen name, a variable name, and, optionally, a default value. Like variables, component parameters can contain different types of data. However, the datatypes supported by component parameters are not quite the same as those supported by variables. Component parameters can be *Arrays*, *Objects*, *Lists*, *Strings*, *Numbers*, *Booleans*, *Font Names*, and *Colors*. These differ from the datatypes supported by variables in two ways:

- Component parameters support three interface-only types: *List*, *Font Name*, and *Color*. A *List* is used to limit the assignment of a parameter's value to one of a predetermined set of options. For example, a parameter named `difficulty` may have its value set from the list: `"hard"`, `"normal"`, `"easy"`. A *Font Name* gives the user a font selection menu for picking a font on the local system. The font name can then be used by *TextFormat* objects. A *Color* gives the user a visual color picker used to select an integer RGB value.
- The primitive types `null` and `undefined` cannot be set directly as the value of a component parameter. This is a limitation of the component interface, not the component parameter itself; code *inside* the clip can assign `null` or `undefined` values to a variable initialized as a component parameter.

To give a component parameter an on-screen name, a variable name, and an optional default value, follow these steps:

1. Click the parameter Name, and enter a short string describing the parameter.
2. Click the parameter Variable, and enter the actual variable name for the parameter.
3. Click the parameter Type, and then select one of the following:
    - Default for parameters that can accept either string or numeric values
    - Array for parameters with array values
    - Object for parameters with object values, including points containing x and y coordinates
    - List for parameters with a predetermined set of possible string or numeric values
    - String for string values
    - Number for numeric (either integer or floating-point) values
    - Boolean for `true` or `false` values
    - Font Name to provide a font selection menu for the parameter
    - Color to provide a visual color picker for the parameter
4. Click the parameter Value, and enter the default value if one is required. This value will appear in the component interface as the initial value for the parameter. The manner in which default values are entered depends on the type of parameter:
    - For Default, String, and Number parameters, click the parameter Value and enter the string or number.
    - For Array, Object, and List parameters, click the parameter Value. In the Values dialog box, add, remove, and arrange items using the plus, minus, and arrow buttons. Click OK to accept your settings.
    - For Boolean, Font Name, and Color parameters, click the parameter Value. Use the menu that appears to

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

select a Boolean value, a font name, or a color.

5. To prevent the parameter's Name from being changed by the component user, select the Parameters are Locked in Instances checkbox.
6. In the Component Definition dialog box, click OK to finalize your parameter settings.

16.8.1.3 Removing and reordering standard component parameters

Sometimes you'll want to remove or rearrange your component's parameters.

To remove a component parameter, follow these steps:

1. In the Library, select the component to modify.
2. Select Component Definition from the Library's pop-up Options menu. (The Component Definition dialog box appears.)
3. Select the parameter to remove.
4. Click the minus (-) button.
5. Click OK.

To rearrange component parameters, follow these steps:

1. In the Library, select the component to modify.
2. Select Component Definition from the Library's pop-up Options menu.
3. Select the parameter to move.
4. Click the arrow buttons until the parameter is in the desired location.
5. Click OK.

16.8.2 Building a Component with a Customized Interface

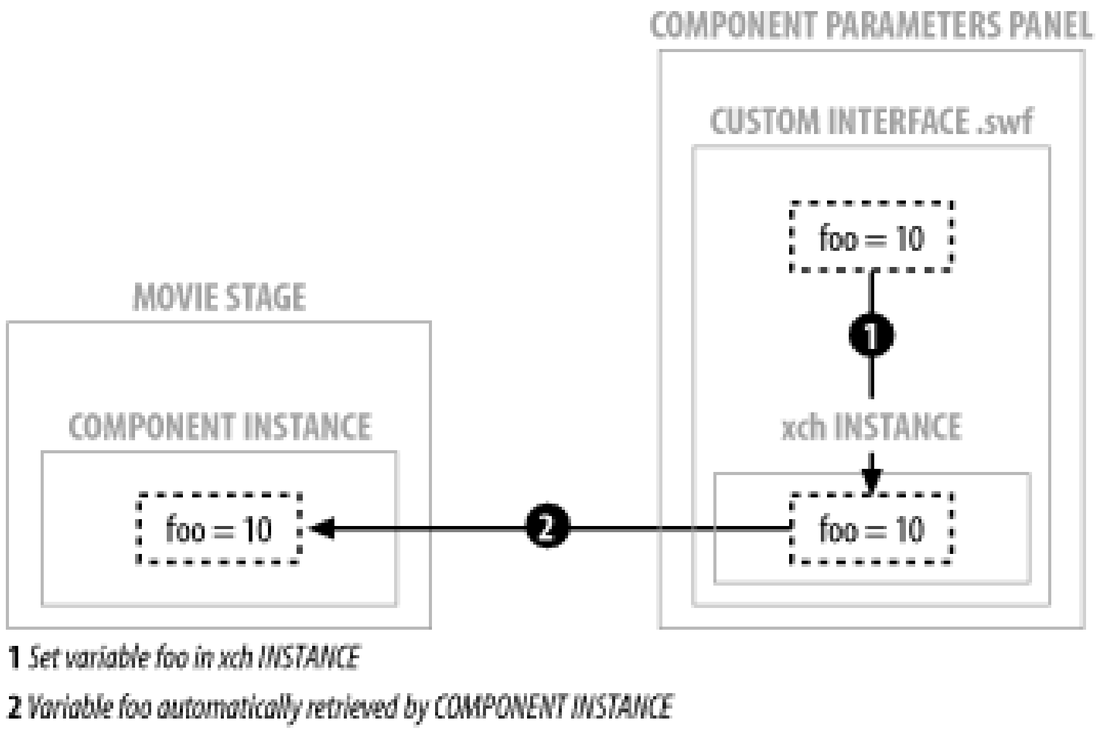
To build a component with a customized interface, we first create a regular movie clip whose behavior is governed by a series of component parameters as described in the preceding section. Next, we create an independent *.swf* file (the so-called *interface swif*) that will be used to set component parameters via custom text boxes, menus, buttons, and other GUI elements. Selected values will automatically be collected and passed to the component as parameters.

The component communicates with the interface *.swf* via the `xch` instance (short for *exchange*), a specially named instance in the interface *.swf*. (We'll see how to create the `xch` instance in a minute.) Figure 16-4 shows how parameter names and values are sent from the interface *.swf* to the component.

Figure 16-4. Custom component communication

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir



Communication between the interface *.swf* and the component occurs in a cycle. When a component instance is selected on stage, the corresponding interface *.swf* loads into the Property inspector or Component Parameters panel. The current parameters in the component instance are then passed to the *.swf* file's `xch` instance. The *.swf* file is expected to retrieve those parameters and set the interface state accordingly. Subsequent variables set in `xch` by the *.swf* file are automatically passed to the component as parameters. When the component instance is unselected, the interface *.swf* is removed from the Component Parameters panel. However, the parameter values are not lost; they are retained by the component instance. Each time the component instance is selected, it passes its parameters back to the *.swf* file's `xch` clip. This cycle allows the interface *.swf* file to stay synchronized with the Component Parameters panel.

The following sections explain how to create the custom interface *.swf* and associate it with a component. For further study, a sample Random Frame component with a customized user interface is available under "Playhead Control" at the online Code Depot.

16.8.2.1 Creating a custom interface .swf file

To create a *.swf* file for use as the custom interface of a component, follow these steps:

1. Create a new Flash document.
2. Create a new layer named *xchLayer*.
3. Select Insert New Symbol to create a blank movie clip symbol.
4. Name the new symbol *xchClip*.
5. On the *xchLayer* layer, place an instance of the *xchClip* symbol.
6. Name the instance `xch`.
7. On frame 1 of the main timeline, create a series of buttons, text fields, and other interface elements used to set variable values (use the built-in Flash UI Components to make this job easier).
8. On frame 2 of the main timeline, add code to read the value of existing component parameters from the `xch` instance. Set the state of the custom interface to reflect the component parameter values. For example, the following code initializes the `param1Input_txt` text field with the value of `param1` from the component:

```
param1Input_txt.text = xch.param1;
```

9. On frame 3 of the main timeline, stop the playhead. Add interface code that sets component parameters as variables in the `xch` instance. Variables in `xch` will automatically be added to the component as parameters. For example, here is how you set values for two custom parameters:

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

```
xch.param1 = value1;  // Values can be of any datatype
xch.param2 = value2;
```

10. Export the *.swf* file.

16.8.2.2 Adding a custom interface to a component

Once we've created the interface *.swf*, we must add it to the component as follows:

1. Close the custom interface *.swf*, and return to the original *.fla* file containing the component.
2. Select the component in the Library.
3. Choose Component Definition from the Library's pop-up Options menu. (The Component Definition dialog box appears.)
4. Next to Custom UI, click Set.
5. In the Custom UI box, for Type, select one of the following:

    - To embed the interface *.swf* file directly into the component *.fla* file, select "Custom UI with *.swf* file embedded in *.fla* file." Embedded interface *.swf* files are stored inside the component *.fla* file, allowing the component *.fla* file to be used and moved independently, without relying on an interface *.swf* file. If an embedded interface *.swf* file changes, it must be updated manually via the Update button in the Custom UI box.

    - To link externally to the interface *.swf* file, select "Custom UI in external *.swf* file." Externally linked interface *.swf* files must always be available at the specified "Custom UI *.swf* file" location and are updated automatically each time a component instance is selected.
6. For Display, select one of the following:

    - If the interface *.swf* file is small enough to fit in the Property inspector, select Display in Property Inspector.

    - If the interface *.swf* file is too large to fit in the Property inspector, select Display in Component Parameters Panel.
7. For a Custom UI *.swf* file, enter the location of the *.swf* file to use as the custom interface, relative to the current *.fla* file. (You can also select the *.swf* file using the Browse button.)

16.8.3 Professional Component Distribution Features

For advanced developers and third parties distributing custom code libraries as components, Flash MX provides a series of features designed to give components a professional polish. These features, outlined next, are described in detail in the article, "Creating Components in Macromedia Flash MX," available at:

http://www.macromedia.com/support/flash/applications/creating_comps/

The features are also discussed in Macromedia's updated Flash MX documentation, available at:

http://www.macromedia.com/support/flash/ts/documents/doc_update_mx.htm

Component description

A component description can be added to the Flash MX Reference panel and the Actions panel Toolbox via plain text or an XML file.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Custom component icon

A component's default icon can be changed to a preset icon (available in the Component Definition box) or a custom icon (supplied as a BMP or PNG file in the component *.fla* file Library). The icon appears next to the component in the Library, Components panel, and Movie Explorer.

Components panel tooltip

A tooltip can be added to components displayed in the Components panel. A tooltip offers a one-line description of the component when the user points to it with the mouse.

Component live preview

A representation of a component (called a *live preview*) can be displayed on stage to give a component user a general idea of what a component looks like at authoring time. A live preview is implemented as a *.swf* file, much like a custom UI. In Flash MX, live previews can be updated once per second.

Components panel installation

A component can be installed into the Flash MX authoring tool so that it is available to any new document via the Components panel. Installation involves either manually dragging the component *.fla* file to the authoring tool's *FlashMX\Configuration\Components* folder or using the Macromedia Extension Manager available at http://www.macromedia.com/exchange/flash.

16.8.4 Using Components

Once component parameters have been assigned to a movie clip, the movie clip officially becomes a "component." By default, components are identified with a special icon:

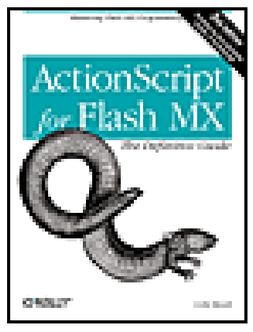
Components appear in either the Library or the Components panel (if set to display in the Components panel).

To use a component instance in a movie, follow these steps:

1. Drag the component from the Library or the Components panel onto the Stage.
2. To set the component's parameters, open either the Property inspector (Window Properties) or the Component Parameters panel (Window Component Parameters), which is used only for components with custom interfaces that will not fit in the Property inspector.
3. If the clip has a standard interface, set each parameter value as follows:
    a. For Default, String, and Number parameters, click the parameter value and enter the string or number.
    b. For Boolean parameters, click the parameter value and select either `true` or `false`.
    c. For Array parameters, double-click the parameter value. In the Values dialog box, click each array element value and enter a string or number. Click OK to accept your array element values.
    d. For Object parameters, double-click the parameter value. In the Values dialog box, click each object property value and enter a string or a number. Click OK to accept your Object properties.
    e. For List parameters, click the parameter value and select an option.
    f. For Color parameters, click the parameter value and select a color from the color palette, or simply enter the integer (hex or decimal) RGB color value.
    g. For Font Name parameters, click the parameter value and select a font from the pull-down menu.
4. If the component has a custom interface, use the tools provided in the custom interface to set the component's parameters.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir



ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Chapter 16. ActionScript Authoring Environment

16.9 Onward!

Well, we're finished our tour of ActionScript fundamentals. It's time to put all your learning to work in a sample application. In Chapter 17 we'll integrate ActionScript with a simple server application to produce a Flash form.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Part I: ActionScript Fundamentals

Chapter 17. Building a Flash Form

Like HTML, Flash supports fill-in forms that can transfer user input to a server-side application or script. In this chapter, we'll see how to capture user input and use the Flash MX *LoadVars* class to communicate with a server-side Perl script. You can use these techniques to save high scores for a video game or retrieve stock ticker data from a server.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Chapter 17. Building a Flash Form

17.1 The Flash Form Data Cycle

Before we delve into the particulars, let's take a macroscopic look at the typical steps in a form submission process:

1. Flash receives data as user input.
2. Flash prepares data for submission to the web server (collects and validates variables).
3. Flash sends data to the web server via HTTP (or, optionally, HTTPS).
4. The web server receives data and passes it to a server-side data-handling application (e.g., Perl script, PHP script, Cold Fusion, or ASP).
5. The data-handling application parses and processes the submitted data.
6. The data-handling application passes the results to the web server, which sends the results to Flash.
7. Flash stores and, optionally, displays the results.

Therefore, a functioning Flash form requires:

- A front end (what the user sees)
- Some Flash scripting that submits the form's contents to a server-side script or application
- A server-side script or application
- Some Flash scripting that handles data returned from the server

Let's examine these in a little more detail.

17.1.1 Flash Client-Side User Input

To supply text-based input, users enter text into on-screen text fields. In ActionScript, an on-screen text field corresponds to an instance of the *TextField* class, which we use to access the field's data. Users can also provide input using interface elements such as pull-down menus, radio buttons, and checkboxes. Prior to Flash MX, these typical form elements were created from scratch by the developer. Flash MX introduced prebuilt form widgets called the *Flash UI Components*: CheckBox, ComboBox, ListBox, PushButton, RadioButton, ScrollBar, and ScrollPane. These components are not a native part of ActionScript; they are sophisticated movie clips that come with the Flash MX authoring tool. To learn how to use the Flash MX UI Components, see Help Tutorials Introduction to Components Tutorial. For a summary of the Flash UI Components' properties and methods, see Appendix G.

Once a user's input has been collected, it should be validated before it's sent to the server. This ensures that our server-side data-handling application always receives usable data. Common validations include checking that all required fields have been filled out and verifying that the correct type of data has been entered. For example, an email address should include a name, followed by an @ symbol, followed by a domain name.

17.1.2 Transmitting Data for Server-Side Processing

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Once the input data is validated, we can safely pass it to the web server. ActionScript provides several tools for transferring form-based data to a web server:

- The *LoadVars* class, described later in this chapter
- The *loadVariables( )* function (used in Flash 4 and Flash 5, but superceded by *LoadVars* in Flash MX)
- The *XML* class's *send( )* and *sendAndLoad( )* methods, described in the *Language Reference*
- The *getURL( )* global function, also described in the ActionScript Language Reference

The web server passes the Flash data to a server-side application (e.g., Macromedia ColdFusion or Microsoft ASP) or a CGI script (e.g., a Perl or PHP script, or a Java servlet).

When describing the web client/server data cycle, we make a point of distinguishing between the web server and a data-handling application. Often, this distinction is implicit—the client always has to make an HTTP request in order to send data to the data-handling application, so it naturally follows that a web server is involved. In Flash form development, however, we must remain aware of the invisible handoff between the web server and the data-handling application. Data moves from Flash to the server either on the end of a URL (using GET) or in a stream of variable names and values (using POST). When a web server error is encountered, Flash does not display the HTTP error messages that the server sends (as a browser would). For example, if the web server can't find a CGI script, it sends a "404 Not Found" message, but Flash doesn't display this message. Similarly, if a CGI script's permissions aren't set correctly, Flash doesn't display an execution-failure error message. In order to isolate client/server problems when working with Flash, monitor the web server's HTTP error log while attempting to run scripts. The web server log will tell you things that Flash won't. For information on problems associated with the HTTP POST method in Flash, see Appendix F.

17.1.3 The Data-Handling Application

Upon receiving data, the data-handling application must *parse* it (i.e., it must interpret the data intelligently and split it into manageable pieces). The parsed data can be manipulated in endless ways by the server application, such as saving it to a database or flat text file for future retrieval.

Once data processing is complete, the data-handling application produces a result to pass back to Flash. The result can be a simple confirmation message (e.g., "Thank you for submitting your information"), a list of records from a database, or the current price of a product.

The application passes the result to the web server, which forwards it on to Flash for storage or display.

> If you're writing a data-handling application, you must set the MIME type of the result to "application/x-www-form-urlencoded". If the MIME type is missing or incorrect, Flash can't interpret the result properly.

17.1.3.1 Sending international characters to Flash

If the response from your server application uses characters above Unicode 127 (i.e., high ASCII characters or characters outside Latin 1), you should use UTF-8-encoded XML to send your data. It is also possible to encode a stream of variables in UTF-8 for use by the *LoadVars* class, but not all server-side applications can specify that encoding. As a last resort, if you know that the server's *code page* (i.e., its primary character set) will definitely match the code page of the Flash client system, you can use `System.useCodepage` to force Flash Player 6 to interpret

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

characters using the system's current code page rather than Unicode (this is, in fact, Flash 5's only supported behavior). For more information, see Section 4.5.1 and `System.useCodepage` in the *Language Reference*.

17.1.4 Flash Receives and Interprets Results

We're nearing the end of the Flash form cycle, but we're not done yet. Consider a movie that looks up stock prices. The user enters a stock ticker symbol and clicks the Get Stock Price button. Before the price can be displayed, the stock-retrieval application must identify and return the share price. While the movie is waiting, you should display a "Loading" message. When the share price is received (i.e., once the data finishes loading), Flash triggers the target *LoadVars* object's *onLoad( )* handler, where we can interpret the data and display it to the user. (Data received via *LoadVars.sendAndLoad( )* is stored in the target *LoadVars* object.)

The Flash form cycle is complete, and we're free to do whatever we like with our precious, well-traveled bytes of content. Now let's put our knowledge to practice by creating a simple fill-in form.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Chapter 17. Building a Flash Form

17.2 Creating a Flash Fill-in Form

This example demonstrates how to send a single text field variable from Flash to a Perl script, named *echo.pl* , and how to rec a response from *echo.pl* in Flash. The movie we'll build consists of an interface and two functions—*initForm ( )* and *submitF ( )* . The interface includes an input text field, a Submit button, and a movie clip that displays *echo.pl* 's response. The *initForm* function creates the server communication objects that send our variable between Flash and *echo.pl* . The *submitForm ( )* fun formats the data and sends it to *echo.pl* . For simplicity, our code is attached to frame 1 of the *scripts* layer. In practice, you should attach the code to a frame following your preloader.

The example files are available at the online Code Depot.

17.2.1 Building the Interface

HTML uses tags such as `<SELECT>` , `<OPTION>` , and `<INPUT>` to create a form's interface. In Flash, a form's interface be built manually in the authoring tool (as in this example) or at runtime via ActionScript. Our form's text field and button are equivalent to HTML's `<INPUT TYPE="TEXT">` , `<INPUT TYPE="TEXTAREA">` , and `<INPUT TYPE="SUBMIT` tags.

> Flash MX offers sophisticated form widgets under the Components panel (Window Components). For information on instantiating components manually and with ActionScript, see Help Tutorials Introduction to Components Tutorial. For information on creating your own components, see Chapter 14 and Chapter 16.

Start building the interface by creating three layers: *scripts* , *input form* , and *output* .

Next, place a user-input text field on the *input form* layer, as follows:

1. Select the Text tool from the Tools panel.
2. On the *input form* layer, drag a rectangle big enough for a user to enter a single line of text.
3. Open the Property inspector (Window Properties).
4. In the Property inspector, for Text Type, select Input Text.
5. In the Property inspector, for Line Display, select Single Line.
6. In the Property inspector, for Instance Name, enter **`input_txt`** . (The `_txt` suffix activates code hinting in the Actions panel.)
7. In the Property inspector, select the Show Border Around Text button.

Alternatively, Flash MX users can create the `input_txt` text field at runtime with the *MovieClip.createTextField( )* metho as follows:

```
// Create and place the text field on the main timeline. Parameters are:
// instance name, depth, x position, y position, width, height
_root.createTextField("input_txt", 1, 250, 235, 150, 20);
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

```
// Make the field an input text field
input_txt.type = "input";

// Set line display to single line
input_txt.multiline = false;

// Enable border and background
input_txt.border = true;
input_txt.background = true;

// Optionally, set colors for the border and background
input_txt.borderColor = 0x666666;
input_txt.backgroundColor = 0x999999;
```

Next, add a Submit button to the *input form* layer as follows:

1. Choose Window → Common Libraries Buttons.
2. From the *Buttons.fla* library's Push Buttons folder, drag an instance of the *push button - blue* button onto the Stage.

To make the Submit button call our custom function *submitForm ( )* , which we'll build later, follow these steps:

1. Select the *push button - blue* button instance.
2. Choose Window Actions.
3. Enter the following code in the Actions panel:

```
on (release, keyPress "<Enter>") {
  submitForm( );
}
```

Flash MX allows us to achieve a similar result by assigning *submitForm ( )* as the callback function for the button's *onRelease* handler. To do so, assign an instance name, such as `submit_btn` , to the button in the Property inspector. Then attach the following code to the main timeline (most suitably in *initForm ( )*):

```
submit_btn.onRelease = submitForm;
```

In most cases, this syntax is preferable, because it keeps the code separated from the physical button on stage. However, we w our form to respond to the Enter key, so we use the Flash 5-style *on(keyPress "<ENTER>")* button handler, which provides a easy way to capture the Enter key. In more complex cases (for example, when handling multiple Submit buttons), we must re to more advanced key handlers. See *Key.addListener ( )* , *Key.onKeyDown ( )* , and *Key.onKeyUp ( )* in the *Language Refer* for details.

To complete the form's front end, build a movie clip, named `response` , to display the results sent by *echo.pl* . The `response` clip has three states, represented by three keyframes labeled `idle` , `loading` , and `doneLoading` :

- When the movie loads, `response` displays the `idle` frame (ordinarily invisible to the user), in which `response` waits for data to begin loading.
- When variables are submitted, `response` displays the `loading` frame, indicating that data has been submitted to tl server and that Flash is waiting for a reply.
- When variables are received, `response` displays the `doneLoading` frame, in which the results are displayed to th user via a text field, `output_txt` .

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

To build `response` , first create a new movie clip symbol:

1. Choose Insert → New Symbol, which opens the Create New Symbol dialog box.
2. In the Name box, enter **`responseClip`** .
3. For Behavior, choose Movie Clip.
4. Click OK.

Next, create an instance of *responseClip* :

1. From the Library, drag an instance of *responseClip* onto the main Stage's *output* layer.
2. Name the *responseClip* instance `response` .

Finally, create the labeled "state" frames in the *responseClip* symbol:

1. In the Library, double-click the *responseClip* symbol (this edits the clip).
2. Create four timeline layers named, from top to bottom, *scripts* , *labels* , *loading* , and *outputField* .
3. Create three keyframes on each layer in frames 1, 2, and 3.
4. On the *labels* layer, label keyframes 1, 2, and 3 as `idle` , `loading` , and `doneLoading` .
5. On the *scripts* layer, at frame 1, add the following code: **`stop( );`** .
6. On the *loading* layer, at frame 2, add the static text, **`"loading, please wait"`** .
7. On the *outputField* layer, at frame 3, use the Text tool to draw a text box.
8. In the Property inspector, for Text Type, select Dynamic Text.
9. In the Property inspector, for Line Display, select Multiline.
10. In the Property inspector, for Instance Name, enter **`output_txt`** .

17.2.2 Sending Data to the Server

To send the value of `input_txt` to *echo.pl* , we'll create the custom functions *initForm ( )* , shown in Example 17-1, and *submitForm ( )* , shown in Example 17-2. The *initForm ( )* function creates one instance of the *LoadVars* class to send our da (`dataSender`) and another to receive *echo.pl* 's response (`dataReceiver`). The *initForm( )* function also assigns a callback function to `dataReceiver` 's *onLoad( )* handler, which executes when *echo.pl* 's reply arrives. The callback func displays the reply in the `response` clip. Notice that *initForm( )* creates a global variable used to refer to the form's timeline Hence, all our references to the `response` clip are fully qualified.

Example 17-1. The initForm( ) function

```
/*
 * Function: initForm( )
 *   Desc: Creates LoadVars objects for sending and
 *         receiving data to and from the server.
 *   Parameters: None.
 */
function initForm ( ) {
  // Create a reference to the form timeline.
  _global.form_mc = this;

  // Create the LoadVars objects.
  // One to send...
  dataSender = new LoadVars( );
  // ...and one to receive.
  dataReceiver = new LoadVars( );
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

```
  // Assign a callback function to the dataReceiver's onLoad handler.
  // It will process the variables when they arrive.
  dataReceiver.onLoad = function ( ) {
    // Prepare the response clip for data display.
    form_mc.response.gotoAndStop("doneLoading");

    // Display the value of output, the loaded variable, in the output_txt fie
    form_mc.response.output_txt.text = this.output;
  }
}
```

The *submitForm( )* function, shown in Example 17-2, gathers the data and transmits it to *echo.pl* .

Example 17-2. The submitForm( ) function

```
/*
 * Function: submitForm( )
 *   Desc: Retrieves user input and sends it to the server.
 *   Parameters: None.
 */
function submitForm ( ) {
  // Assign the value of input_txt to a property of dataSender. The escape( )
  // function encodes the & character to avoid network transfer errors.
  dataSender.formInput = escape(input_txt.text);

  // Display a loading message in the response clip.
  form_mc.response.gotoAndStop("loading");

  // Display what we're sending in the Output window, for debugging purposes.
  trace("Sending " + dataSender.toString( ));

  // Send the properties of dataSender to the server, and wait for a reply.
  dataSender.sendAndLoad("http://www.yourserver.com/cgi-bin/echo.pl",
                         dataReceiver, "GET");
}
```

The `formInput` property of the `dataSender` object contains the data to be transmitted—the current input of the `input_txt` text field, escaped (encoded) for safe network transfer:

```
dataSender.formInput = escape(input_txt.text);
```

While the user waits for *echo.pl* to respond to our form submission, the code displays a loading message in the `response` c

```
form_mc.response.gotoAndStop("loading");
```

For testing purposes, we display the data to be sent in the Output window, using the *toString ( )* method:

```
trace("Sending " + dataSender.toString( ));
```

Finally, we submit the form using *sendAndLoad ( )* , instead of *send ( )* or *load ( )* , because we're both sending variables to, a receiving variables from, *echo.pl* . The *sendAndLoad ( )* method sends `formInput` (and any other `dataSender` propert to *echo.pl* :

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

```
dataSender.sendAndLoad("http://www.yourserver.com/cgi-bin/echo.pl",
                       dataReceiver, "GET");
```

The *sendAndLoad ( )* method takes three parameters: `url` (the location of the server script), `targetObject` (the *LoadV* object that should receive the results), and `method` (the HTTP method, either "GET" or "POST"). When *sendAndLoad ( )* executes, `dataSender` 's properties are converted to a single ampersand-delimited string of variable name/value pairs and to *echo.pl* .

17.2.3 The Perl Script, echo.pl

When the user clicks the Submit button in our form, Flash initiates an HTTP GET request that executes the Perl script *echo.p* For our form to work, *echo.pl* must be placed in a CGI-enabled directory of a web server and configured by the server's administrator as follows:

- The script must be executable (typically, this means setting the script file's permissions to 755).
- On Unix, the path to the Perl interpreter must be set in the script.

Example 17-3 shows the source code for *echo.pl* . Note that in Perl, the # character indicates a comment.

Example 17-3. The source code for echo.pl

```
#! /usr/local/bin/perl
#-------------------------------------------------------------------------
#  Name:      Simple Flash Echo
#  Version:   1.2.1
#  Author:    Derek Clayton  derek@moock.org
#  Description:  Echoes back name/value pairs received from a Flash GET or POS
#-------------------------------------------------------------------------
# MAIN
#-------------------------------------------------------------------------
use CGI;                        # Use the CGI.pm for easy parsing
$query = new CGI;               # Query object
$echoString = "output=";        # Initialize our output string
&getInput;                      # Get the input received from Flash
&writeResponse;                 # Write the response back to Flash
exit;                           # Exit the script
#-------------------------------------------------------------------------
sub getInput {
  # For each key get the associated value and add to the echo string
  foreach $key ($query->param) {
    $value = $query->param($key);
    $echoString .= "$key:$value\n";
  }
  # Remove the trailing newline (\n) before writing response
  chomp($echoString);
}

sub writeResponse {
  # Set content type for Flash
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

```
  print $query->header({"Content-Type"=>"application/x-www-form-urlencoded"});
  # Write the output
  print $echoString;
}
```

The *echo.pl* script simply returns a single variable, `output` , that contains a single string representing all the variables we se the server. Here are the specific tasks *echo.pl* performs.

1. It accepts data sent by Flash and parses that data into a series of variable names and values. For example, it converts ou `dataSender.formInput` property to the variable `formInput` . Any other properties attached to `dataSend` would likewise be converted to variables.
2. It assembles those variable names and values into a single string (to return to Flash) of the form:

   `output=`*`name1`*`:`*`value1`*`\n`*`name2`*`:`*`value2`*`\n...`*`namen`*`:`*`valuen`*

   For example, in response to our form submission, *echo.pl* creates the string:

   `output=formInput:`*`theInputValue`*

   If we give `dataSender` another property, `country` , with a value of "Canada", then *echo.pl* creates the strin

   `output=formInput:`*`theInputValue`*`\ncountry:Canada`

3. It returns the string to Flash.

When Flash receives the string from *echo.pl* , it finds the `output` variable and automatically adds a corresponding `output` property on `dataReceiver` . The value of the `output` property is the string containing the list of variables we sent to `echo.pl` .

Obviously, *echo.pl* is merely a proof of concept. When applied, however, the concept can have interesting and powerful resul For an example of a more fully developed Perl system, see the flat file database sample available under "Server Communicati at the online Code Depot.

17.2.4 Receiving Results from the Server

Recall that when we sent the properties of `dataSender` to *echo.pl* , we requested that *echo.pl* 's return value be stored in t `dataReceiver` object:

```
dataSender.sendAndLoad("http://www.yourserver.com/cgi-bin/echo.pl",
                       dataReceiver, "GET");
```

When data is received from *echo.pl* , Flash executes the callback assigned to *dataReceiver.onLoad( )* . We originally assigned callback inside our *initForm( )* function in Example 17-1; let's take a closer look at it now:

```
dataReceiver.onLoad = function ( ) {
  // Prepare the response clip for data display.
  form_mc.response.gotoAndStop("doneLoading");

  // Display the value of output, the loaded variable, in the output_txt field
  form_mc.response.output_txt.text = this.output;
}
```

This function moves `response` 's playhead to the `doneLoading` frame, where the text field `output_txt` resides. Th

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

it displays the value of `output` (supplied by *echo.pl*) in the `response` clip's `output_txt` text field. Note that the call is executed as a method of `dataReceiver` , so the `dataReceiver` object is referred to as `this` inside the function.

All that's left to do is try the form out! Export your movie, enter some text in the `input` text field, and click the Submit butt If you entered "test" you should see "formInput:test" appear in the `output_txt` text field. If the form doesn't work the firs time, make sure your server script is configured properly. And remember, you can study the functional version posted at the online Code Depot.

Reader Exercise: Add a Reset button that clears the value of the form's input field. Simply create a new button in the `formC` and attach the following code to it:

```
on (release) {
  input_txt.text = "";
}
```

Also, try adding more than one input field to your form; the Perl script will faithfully return as many variables as you send it, assigned as a group to the variable `output` . Can you split them up, using the *String* class's methods, and display each varia value separately?

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Chapter 17. Building a Flash Form

17.3 Onward!

Well, I'd say our work in this chapter wasn't too shabby. Think of all the powerful things you can do with Flash forms, such as remembering a user's preferences, providing a discussion forum, or loading dynamic product information for a catalog. The flat file Perl database at the online Code Depot will take you further down the road toward building those kinds of applications.

Our ActionScript conversation is over, but yours has just begun. By reading Part I, you've learned to speak ActionScript—now it's time to apply that knowledge to your own projects. Before you embark, here are a few parting thoughts:

- As with any art form, learning to program is a process, not an event. For as long as you program, you'll learn more about programming. Consider the multiple-choice quiz from earlier—see Chapter 1 (Examples Example 1-1 through Example 1-4), Chapter 9 (Example 9-13), Chapter 11 (Example 11-7), and Chapter 12 (Tables Table 12-2 through Table 12-5)—we rebuilt it four times! Each time we refined our approach, added features, and learned something we hadn't considered before. Just as each painting teaches the painter something new about her subject matter, so creating and recreating applications will reveal new approaches to you.

- There's practical help in Part II, the ActionScript Language Reference, which contains detailed descriptions of ActionScript's built-in functions, properties, classes, and objects. While you may not want to read it from start to finish, you should definitely skim each topic to get a sense of ActionScript's capabilities.

- Revisit this book, including the *Language Reference*, often. You'll pick up new insights each time through, because you'll be considering the information with a higher level of understanding and will be able to relate concepts to real-world experiences. Treat the ActionScript Language Reference as a dictionary and keep it by your side while you work.

- There's a thriving community of Flash developers offering ideas and solutions and—most importantly—sharing source code! Dissect as much as you can. Identify the things you can't understand and look those topics up in this book. Macromedia maintains a list of web sites and mailing lists devoted to Flash at http://www.macromedia.com/support/flash/ts/documents/flash_websites.htm.

- Don't limit your exploration of ActionScript to this book. Look at things from multiple angles by consulting other sources of knowledge and inspiration, such as those listed in Appendix A and the Preface. You'll also find a long list of Flash resources at http://www.moock.org/moockmarks and reviews of worthwhile Flash books at http://www.moock.org/webdesign/books.

- Finally, remember to drop by the support site for *ActionScript for Flash MX: The Definitive Guide*, http://www.moock.org/asdg, for lots of code samples, technotes, and discussions of new topics.

With that, I wish you happy coding! Throw an extra iteration into a *while* loop for me sometime. :)

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Part II: Language Reference

This *Language Reference* explains how to use each built-in class, object, function, property, event handler, directive, and pragma supported by ActionScript. You'll use this *Language Reference* regularly to accomplish specific tasks in your daily programming work.

- ActionScript Language Reference

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Part II: Language Reference

Chapter 18. ActionScript Language Reference

This *Language Reference* documents all the supported classes and objects of ActionScript, explaining their general purpose, usage, properties, methods, and event handlers. Also covered are the *global functions* and *global properties* (those not attached to a class or an object, but available in standalone form, throughout a movie).

All entries in the *Language Reference* are alphabetized regardless of whether an entry is a property, method, global function, or event handler. For example, the global function *duplicateMovieClip( )* is listed under "D" after the *Date* class (not, say, in a separate "Global Functions" section). However, each entry's title clearly indicates the type of item being described so that there can be no confusion as to the item's purpose or usage. Furthermore, many items are alphabetized by the object or class to which they belong. For example, *duplicateMovieClip( )* comes in both a global function and movie clip method form, so you'll find it listed separately as *duplicateMovieClip( ) Global Function* under "D" and as *MovieClip.duplicateMovieClip( ) Method* under "M." Some operators, such as *delete* and *typeof*, are included in the *Language Reference*, but for complete coverage of operators (including arithmetic and logical operators) see Chapter 5. Likewise, statements and loops, such as *if* and *for*, are covered in Chapter 6, Chapter 7, and Chapter 8. If you're unsure of where to find an item, consult the Index at the back of the book.

All *Language Reference* entries have been updated to reflect additions and changes implemented in Flash MX and Flash Player 6 (for the sake of brevity, we refer to the Flash MX authoring tool and the Flash Player 6 browser plugin collectively as "Flash 6"). Compatibility information is provided for previous versions. New objects and classes have been added; new methods and properties have also been added to existing classes. So, you should skim the entire *Language Reference* to familiarize yourself with the current capabilities of the language. In some cases, the entries for items supported in Flash 5 have been augmented with better examples, even if the changes were not directly related to Flash 6.

For an explanation of the formatting conventions used in this *Language Reference*, see Section P.10 in the Preface.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Chapter 18. ActionScript Language Reference

18.1 Global Functions

Global functions are built-in functions that are available throughout an entire movie—they can be invoked from any frame, button, or event handler in a movie (unlike methods, which must be invoked using a reference to a specific object).

Table 18-1 lists the global functions available in Flash 6, including two new global functions: *clearInterval( )* and *setInterval( )*. All other global functions listed in the table are available in Flash 5 as well. Table 18-1 also lists the directives and pragmas, which begin with the # character — namely, `#include`, `#initclip`, and `#endinitclip`.

Table 18-1. ActionScript global functions

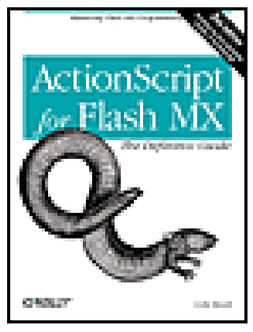
Boolean( )	*gotoAndStop( )*	*parseInt( )*	*stopAllSounds( )*
call( )	`#include`	*play( )*	*stopDrag( )*
clearInterval( )[1]	`#initclip`[1]	*prevFrame( )*	*String( )*
Date( )	*int( )*[2]	*prevScene( )*	*targetPath( )*
duplicateMovieClip( )	*isFinite( )*	*print( )*	*tellTarget( )*[2]
`#endinitclip`[1]	*isNaN( )*	*printAsBitmap( )*	*toggleHighQuality( )*[2]
escape( )	*loadMovie( )*	*printAsBitmapNum( )*	*trace( )*
eval( )	*loadMovieNum( )*	*printNum( )*	*unescape( )*
fscommand( )	*loadVariables( )*	*random( )*[2]	*unloadMovie( )*
getProperty( )	*loadVariablesNum( )*	*removeMovieClip( )*	*unloadMovieNum( )*
getTimer( )	*nextFrame( )*	*setInterval( )*[1]	*updateAfterEvent( )*
getURL( )	*nextScene( )*	*setProperty( )*	
getVersion( )	*Number( )*	*startDrag( )*	
gotoAndPlay( )	*parseFloat( )*	*stop( )*	

[1] Supported in Flash 6, but not in prior versions.

[2] Deprecated in Flash 5 and later versions.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir



ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Chapter 18. ActionScript Language Reference

18.2 Global Properties

Global properties, like global functions, can be accessed from any script in a movie. They store information that is generally useful from any point in our code. Many of them affect the entire Flash Player, not just a particular movie clip or movie.

Table 18-2 lists the global properties available in Flash 6, including the new global property `_global`. All other global properties listed in the table are available in Flash 5 as well.

Table 18-2. ActionScript global properties

Property name	Description
`_focusrect`	The highlight state of buttons activated via the keyboard
`_global`[3]	A reference to the global object
`_highquality`[4]	The rendering quality of the Player
`Infinity`	A constant representing the Infinity value of the *number* type
`-Infinity`	A constant representing the -Infinity value of the *number* type
`_level`*n*	A document level in the Player, such as `_level0`
`NaN`	Not-A-Number; the value of the *number* type representing invalid numeric data
`_quality`	The rendering quality of the Player
`_root`	A reference to the main movie timeline of the current level
`_soundbuftime`	Number of seconds of a streaming sound to preload
`$version`[4]	The version of the Flash Player

[3] Supported in Flash 6, but not in prior versions.

[4] Deprecated in Flash 5 and later versions.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Chapter 18. ActionScript Language Reference

18.3 Built-in Classes and Objects

This *Language Reference* assumes that you understand the terms *class*, *object*, and *instance*, as discussed exhaustively in Chapter 12. The built-in classes of ActionScript are used to create objects that can control a movie and manipulate data. The built-in ActionScript classes added in Flash MX include:

Button
Function
LocalConnection
LoadVars
TextField
TextFormat

They join the *Array*, *Boolean*, *Button*, *Color*, *Date*, *MovieClip*, *Number*, *Object*, *Sound*, *String*, *XML*, *XMLnode*, and *XMLSocket* classes, which were also supported in Flash 5.

> The *MovieClip* class is enhanced substantially in Flash 6. Most notably, an entire Drawing API and support for button events have been added.

To create an instance of a particular class, we use the class's *constructor* function with the *new* operator. For example, to make a new object of the *Color* class, we use the *Color* constructor as follows:

```
theColor = new Color(_root);
```

For each class entry in the *Language Reference*, the *Constructor* subheading shows how to create new objects of that particular class (i.e., it demonstrates each class's constructor function syntax). The class entry also includes a summary of the properties, methods, and event handlers available for objects in the class, along with a description of the class's purpose and typical use. Some classes also define methods or properties that are accessed through the class constructor itself, not individual instances. These methods and properties are listed as *Class Methods* and *Class Properties*. Full details for each class's properties, methods, and event handlers are given in an alphabetical list after each class's general introduction.

The built-in ActionScript objects added in Flash MX include:

Accessibility
Capabilities
SharedObject
Stage
System

They join the *Arguments*, *Key*, *Math*, *Mouse* and *Selection* objects, which were also supported in Flash 5. (See the

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Preface for notes on the *CustomActions* and *LivePreview* objects, which were added in Flash MX but are not covered in this *Language Reference*.) Objects are interspersed alphabetically with the class descriptions in this *Language Reference* but are distinguished by the word *Object* (e.g., *Math Object*). Unlike true classes, which use a constructor function to instantiate multiple objects, these standalone objects are never instantiated (i.e., they are not constructed with the *new* operator); rather, they are predefined objects that exist solely to collect related functionality into a single package. The *Math* object, for example, provides convenient access to common mathematical functions and constants.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

18.4 Entry Headings

Each entry in the *Language Reference* is presented in the following basic format:

Entry Name **Availability**

Purpose ***Access***

`Synopsis`

Table 18-3 describes these headings, as well as additional headings used to document each item.

Table 18-3. Language Reference headings

Heading	Description
Availability	Indicates when the item was added to ActionScript (or when it became an Action, if the item predates formal ActionScript). For example, if the *Availability* is listed as "Flash 5," the feature is also supported in Flash 6, unless otherwise noted. Furthermore, if the *Availability* explicitly or implicitly includes Flash 6, the feature is supported both in the Flash MX authoring tool and the Flash Player 6 browser plugin, unless otherwise noted. This heading also summarizes backward-compatibility and deprecation status.
Access	Applies to property entries only. Describes whether the property value can only be retrieved ("read-only") or both retrieved and assigned ("read/write").
Purpose	A brief summary of the item's purpose.
Synopsis	Illustrates the abstract syntax required by the item. Italicized code indicates text that must be replaced by the programmer, such as with a string, integer, expression, or identifier. Some functions can be called with differing types and numbers of arguments, in which case the *Synopsis* may show multiple supported syntaxes. Note that the *Synopsis* does not indicate whether function, method, or constructor arguments are optional; that information is available under the *Arguments* subheading.
Arguments	Applies to method and function entries only. Describes the parameters used with the method or function, as listed in the *Synopsis*.
Operands	Applies to operator entries only. Describes the operands used with the operator, as listed in the *Synopsis*.
Returns	Applies to method and function entries only (properties always "return" the value stored within them). Describes the return value of the method, if any. If the method does not have a return value, the *Returns* subheading is omitted. Functions without documented return values always return `undefined`.
Description	Explains how the item works and how it might be used in a practical scenario.
Usage	Describes noteworthy peculiarities of the item.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Bugs	Describes known problems associated with the item.
Example	Offers sample code showing the item in use, although shorter samples sometimes appear under the *Description* subheading.
See Also	Lists cross-references to related entries within this *Language Reference* and to related material in Part I. Refer also to the book's index and resources listed in the Preface and Appendix A.

Copyright © 2002 O'Reilly & Associates, Inc. All rights reserved.
1005 Gravenstein Highway North
Sebastopol, CA 95472

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Book: ActionScript for Flash MX: The Definitive Guide, 2nd Edition
Section: Chapter 18. ActionScript Language Reference

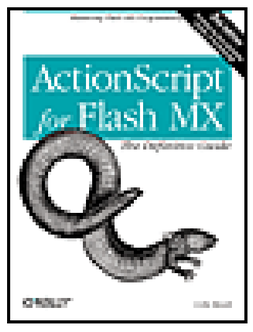
18.5 Alphabetical Language Reference

The following entries document ActionScript's event handlers, methods, and properties of objects and classes, plus global functions and properties. Refer to the Index for the operators and statements covered elsewhere in this book. This reference documents the core ActionScript language. For a summary of the methods and properties of the Flash UI Components included with Flash MX, and an explanation of why they are not documented here, see Appendix G. See Chapter 14 for more information about components, including resources for documentation on the Flash UI Components.

Copyright © 2002 O'Reilly & Associates, Inc. All rights reserved.
1005 Gravenstein Highway North
Sebastopol, CA 95472

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir



Chapter 18. ActionScript Language Reference

Accessibility Object Flash 6 (ActiveX for Windows IE only)

tools for developing advanced accessible UI components

Methods

isActive( )

Indicates whether an accessibility aid program (generally a screen reader) is communicating with the Flash Player.

sendEvent( )

Sends an event notice to Microsoft Active Accessibility (MSAA).

Description

The *Accessibility* object manages communication between the Flash Player and screen readers or accessibility aids such as GW Micro's Window-Eyes (http://www.gwmicro.com). The *isActive( )* method checks whether an accessibility aid is communicating with the Flash Player. It is useful when dynamically generating content in a movie based on the presence of accessibility support. The *sendEvent( )* method allows you to send information about UI component events to accessibility aids and is intended for developers of accessible UI components.

At the time of this writing, the Macromedia ActionScript Component Accessibility (MASCA) API works with Microsoft Active Accessibility (MSAA) only; hence, only the ActiveX version of Flash Player 6 supports communication with an external accessibility application. In the future, more versions of the Flash Player are likely to include support. Use of the MASCA API requires some knowledge of MSAA. To determine whether the current Player has accessibility support, use `capabilities.hasAccessibility`.

Note that *Accessibility* is a predefined object, not a class. Its properties, methods, and event handlers are accessed directly, as in:

```
Accessibility.isActive();
```

Many of the *Accessibility* implementation details have yet to be fully exposed to the developer community. In general, ActionScript accessibility is required only by advanced interface components. For examples, see the Flash UI Components included in the Flash MX authoring tool. As third-party component development proliferates, Macromedia plans to publish accessibility specifications and tutorials.

For more information on accessibility in Flash, see:

http://www.macromedia.com/macromedia/accessibility/features/flash/

downloaded from: lib.ommolketab.ir

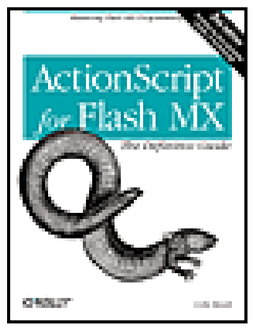
downloaded from: lib.ommolketab.ir

See Also

capabilities.hasAccessibility

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir



Chapter 18. ActionScript Language Reference

Accessibility.isActive( ) Method Flash 6

indicates whether an accessibility aid program (generally a screen reader) is communicating with the Flash Player

```
Accessibility.isActive()
```

Returns

A Boolean: `true` if an accessibility aid is communicating with the Player, or `false` if not.

Description

The *isActive( )* method returns a Boolean indicating whether an external accessibility application, such as a screen reader, is currently being used to read what the Flash Player is displaying. If a screen reader is in use, *isActive( )* returns `true`, otherwise, it returns `false`.

Not all Flash Players support communication with an accessibility application. To check whether the specific Flash Player running the movie supports accessibility, use `capabilities.hasAccessibility`. Currently, only the ActiveX version of Flash Player 6 for Windows Internet Explorer supports accessibility. Players that do not support accessibility, including the Netscape plugin for Windows and the Macintosh plugin, always return `false` for *Accessibility.isActive( )*.

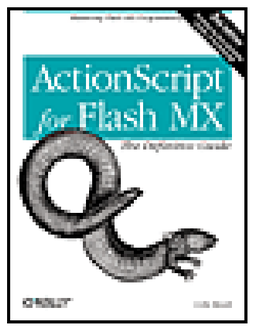
The *isActive( )* method depends on the Flash Player having received an initial communication from an accessibility aid program. There is a small, but not negligible, delay between when the Flash Player starts up and when it receives the first such message. Thus, when *isActive( )* is invoked early in a Flash movie, it may return `false` even though there is actually an accessibility aid running. To work around this problem, build a delay of approximately one second into your movie before calling *isActive( )*.

See Also

Accessibility.sendEvent( ), `capabilities.hasAccessibility`

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir



ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Accessibility.sendEvent( ) Method

Flash 6 (ActiveX for Windows IE only)

---

send an event notice to MSAA

```
Accessibility.sendEvent(movieClip, childID, event, isNonHtml)
```

Arguments

movieClip

The movie clip instance sending the event.

childID

The child interface element to which the event applies (for example, an individual list item in a list box). Use 0 to indicate that the event applies to the specified `movieClip`.

event

A constant indicating the type of event. The event names and values used are a subset of the MSAA event constants listed in Microsoft's MSAA documentation at http://msdn.microsoft.com/at.

isNonHtml

A Boolean indicating whether the event is (`false`) or is not (`true`) one of the standard event types that can be generated by an HTML form. When set to `true`, this argument prevents problems in some screen readers that interpret Flash content in Internet Explorer as part of the HTML page. Defaults to `false`.

Description

The *sendEvent( )* method is used by accessible UI components to notify Microsoft Active Accessibility (MSAA) of an interface change in the component. MSAA passes the message on to a screen reader application that reports the change to the user. For example, a radio button movie clip would use *sendEvent( )* to notify MSAA when its name changes or when it is selected or unselected. If `movieClip` is not a movie clip instance, or if `childID` or `event` are missing, *sendEvent( )* fails silently. If no accessibility aids are running, calling *sendEvent( )* has no effect.

The *sendEvent( )* method is only one part of the MASCA API that, at the time of this writing, is not yet publicly documented, but it is an important part of the company's Flash component strategy. The API requires components to define special objects and constants in order to communicate with MSAA. For example, each component that wishes to be accessible must store an object in the reserved `_accImpl` property. This object, known as the *ActionScript Accessibility Object* (ASAO), implements methods and properties that describe the component in accessibility terms and expose it to MSAA (much like the C++ *IAccessible* interface, for which an ASAO is, in fact, a simplified wrapper).

Once the required objects and constants have been defined according to the MASCA API, a component depicts

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

interface interaction in accessible terms via *sendEvent( )*. For example, the *setLabel( )* method of the radio button component uses *sendEvent( )* to notify a screen reader when the button's label changes:

```
FRadioButtonClass.prototype.setLabel = function(label)
{
  this.fLabel_mc.setLabel( label );
  this.txtFormat();
  // ACCESSIBILITY
  if(Accessibility.isActive()){
    Accessibility.sendEvent( this, 0, this.EVENT_OBJECT_NAMECHANGE );
  }
}
```

For more information on the *IAccessible* interface, see:

http://msdn.microsoft.com/library/en-us/msaa/msaapndx_6qlh.asp

For more information on MSAA, see:

http://msdn.microsoft.com/library/en-us/msaa/msaastart_9w2t.asp

Microsoft regularly changes links on its developer web site. If these documents are no longer available, consult http://msdn.microsoft.com/at.

See Also

`capabilities.hasAccessibility`

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Chapter 18. ActionScript Language Reference

Arguments Object Flash 5; caller property added in Flash 6

access to function arguments, the current function, and the calling function

```
arguments[elem]
arguments.propertyName
```

Properties

callee

A reference to the function being executed.

caller

A reference to the calling function.

length

The number of arguments passed to the function being executed.

Description

The *Arguments* object is stored in the local `arguments` variable of every function and is accessible only while a function is executing. *Arguments* is both an array and an object. As an array, `arguments` stores the values of the parameters passed to the currently executing function. For example, `arguments[0]` is the first passed parameter, `arguments[1]` is the second passed parameter, and so on. All arguments passed are stored in the array, including those arguments not formally declared as parameters. As an object, *Arguments* stores the `callee` and the `caller` properties, which can be used to identify or invoke the current function and the calling function. As of Flash 6, `arguments` supports all *Array* methods and properties (*pop( )*, *shift( )*, etc.); prior to Flash 6, `arguments` supported the `length` property, but not the *Array* methods.

See Also

Section 9.8.2

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Chapter 18. ActionScript Language Reference

arguments.callee Property Flash 5

a reference to the function being executed ***read/write***

`arguments.callee`

Description

The `callee` property stores a reference to the function currently executing. We can use this reference to execute the current function again or to identify the current function through comparison.

Example

```
function someFunction () {
  trace(arguments.callee =  = someFunction);    // Displays: true
}

// An unnamed recursive function
countToTen = function () {
  i++;
  trace(i);
  if (i < 10) {
    arguments.callee();
  }
};

// Invoke the unnamed recursive function
countToTen();
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Chapter 18. ActionScript Language Reference

arguments.caller Property — Flash 6

a reference to the calling function — ***read/write***

`arguments.caller`

Description

The `caller` property stores a reference to the function that called the current function. If the current function was not called by another function, `caller` is `null`. For example:

```
// Create a function that calls two other functions
function formatPage () {
  formatTitle();
  formatBody();
}

function formatTitle () {
  // Check if the function was called by formatPage( )
  trace(arguments.caller =  = formatPage);
}

function formatBody () {
  // Check if the function was called by formatPage( )
  trace(arguments.caller =  = formatPage);
}

// Now call formatTitle( ) and formatBody( ) from formatPage( )

formatPage();
// Displays: true and true

// Then call formatTitle( ) directly
formatTitle();
// arguments.caller will be null. Displays: false
```

We can use `caller` to execute or identify a calling function and to check whether a function was called directly or by another function.

Usage

Note that in Netscape 4, JavaScript's `arguments.caller` refers to the *Arguments* object of the calling function, not the calling function itself. In JavaScript, `arguments.caller` is deprecated as of version 1.3, and it is not

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

supported by Internet Explorer or later versions of Netscape.

Bugs

Macromedia's initial documentation for Flash MX incorrectly claims that `arguments.caller` refers to the calling function's *Arguments* object. It refers, in fact, to the calling function itself, not the caller's *Arguments* object.

Example

The following code shows a function that aborts itself when called directly, ensuring that it is only called by a larger routine:

```
function subFunction () {
  if (arguments.caller =  = null) {
    return "Function cannot be called directly.";
  }
  // proceed with function...
}
```

See Also

`arguments.callee`

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Chapter 18. ActionScript Language Reference

arguments.length Property Flash 5

the number of parameters passed to an argument ***read/write***

`arguments.length`

Description

The `length` property stores an integer representing the number of elements in the `arguments` array, which equates to the number of parameters passed to the currently executing function.

Example

We can use the `length` property of `arguments` to determine whether a function was invoked with the correct number of parameters. The following example checks whether two arguments have been passed to *someFunction( )*. Checking whether the passed arguments are of the correct type is left as an exercise to the reader. (Hint: see the *typeof* operator.) Here's the code:

```
function someFunction (y, z) {
  if (arguments.length != 2) {
    trace("Function invoked with wrong number of parameters");
    return;
  }
  // Proceed with normal function body...
}
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Chapter 18. ActionScript Language Reference

Array Class Flash 5; sortOn( ) method added in Flash 6

support for ordered lists of data

Constructor

```
new Array()
new Array(len)
new Array(element0, element1, element2,...elementn)
```

Arguments

len

A nonnegative integer specifying the size of the new array.

element0, ...elementn

A list of one or more initial values to be assigned as elements of the array.

Properties

length

The number of elements in an array (including empty elements).

Methods

concat( )

Create a new array by appending additional elements to an existing array.

join( )

Convert the elements of an array to strings, and concatenate using a customizable delimiter.

pop( )

Remove and return the last element of an array.

push( )

Add one or more elements to the end of an array.

reverse( )

Reverse the order of elements in an array.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

shift( )

Remove and return the first element of an array.

slice( )

Create a new array using a subset of elements from an existing array.

sort( )

Sort the elements of an array according to the specified rule.

sortOn( )

Sort an array of objects by the specified property.

splice( )

Remove elements from, and/or add elements to, an array.

toString( )

Convert an array to a string of comma-separated element values.

unshift( )

Add one or more elements to the beginning of an array.

Description

We use the properties and methods of the *Array* class to manipulate the data stored in the elements of an *Array* object. See Chapter 11 for exhaustive details on what arrays are and how you can use them, plus detailed definitions of the terminology used in this section. See also the `[ ]` and dot operators, which are used to access array elements and form array literals, as described in Chapter 5.

Usage

If the *Array* constructor is invoked with a single integer argument, that argument is used to set the length of the new array, which is then filled with `undefined` values. If two or more arguments are supplied to the constructor or if a single nonnumeric argument is supplied to the constructor, the arguments are used as the initial values for elements in the array, and the length of the array is determined by the number of arguments specified.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Array.concat( ) Method — Flash 5

create a new array by extending an existing array

`array.concat(`*`value1, value2, value3,...valuen`*`)`

Arguments

value1, ...valuen

A list of expressions to be added to the end of `array` as new elements.

Returns

A new array containing all the elements of `array`, followed by the elements `value1, ...valuen`.

Description

The *concat( )* method returns a new array created by appending new elements to the end of an existing array. The original array is left unchanged. Use the *push( )*, *splice( )*, or *shift( )* method to modify the original array.

If an array is used as an argument to *concat( )*, each element of that array is appended separately. That is, the result of *arrayX.concat(arrayY)* will be an array formed by appending each element of `arrayY` to the end of `arrayX`. The resulting array will have a `length` equal to `arrayY.length + arrayX.length`. Nested arrays, however, are not similarly flattened.

Example

```
// Create an array
myListA = new Array("apples", "oranges");

// Set myListB to ["apples", "oranges", "bananas"]
myListB = myListA.concat("bananas");

// Create another new array
myListC = new Array("grapes", "plums");

// Set myListD to ["apples", "oranges", "bananas", "grapes", "plums"]
myListD = myListB.concat(myListC);

// Set myListA to ["apples", "oranges", "bananas"]
myListA = myListA.concat("bananas");
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

```
// Create an array
settings  = ["on", "off"];
// Append an array containing a nested array
options = settings.concat(["brightness", ["high", "medium", "low"]]);
// Sets options to: ["on", "off", "brightness", ["high", "medium", "low"]]
// not: ["on", "off", "brightness", "high", "medium", "low"]
```

See Also

Array.push( ), *Array.shift( )*, *Array.splice( )*; Section 11.7.3.4

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Array.join( ) Method — Flash 5

convert an array to a string

```
array.join()
array.join(delimiter)
```

Arguments

delimiter

An optional string to be placed between elements in the newly created string. Defaults to a comma if not supplied.

Returns

A string composed of all the elements of `array`, converted to strings and separated by `delimiter`.

Description

The *join( )* method returns a string created by combining all the elements of an array, as follows:

1. Convert each element in the array to a string according to the rules specified in Table 3-2 in Chapter 3.
2. Add `delimiter` to the end of each converted-element string, except the last one.
3. Concatenate the converted-element strings into one long string.

Note that elements that are themselves arrays are converted to strings via the *toString( )* method, so nested array elements are always delimited by commas, not by the delimiter used in the *join( )* invocation.

Example

```
fruit = new Array("apples","oranges","bananas","grapes","plums");
// Set fruitString to "apples,oranges,bananas,grapes,plums"
// (default delimiter is a comma with no following space)
fruitString = fruit.join();
// Use ", " as the delimiter to create nicer formatting than toString( )
// and set fruitString to "apples, oranges, bananas, grapes, plums" (with a
// space after each comma)
fruitString = fruit.join(", ");
```

See Also

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Array.toString( ), *String.split( )*; "The *join( )* Method," in Chapter 11

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Chapter 18. ActionScript Language Reference

Array.length Property — Flash 5

the number of elements in an array — *read/write*

array.length

Description

The length property is a nonnegative integer specifying the number of elements in an array. An array with no elements has a length of 0; an array with 2 elements has a length of 2. Note that the index number of the first element in an array is 0, so length is always one greater than the index of the last element in the array.

The length property of an array indicates how many numbered elements the array currently contains, including empty elements (those containing null or undefined). For example, an array may have values for elements 0, 1, 2, and 9, but elements 3 through 8 may be empty. Such an array has a length of 10, because it has 10 element positions (0 through 9), even though only 4 positions are occupied by useful values.

The length property changes automatically whenever any elements are added or removed via the *Array* class methods, such as *push( )*. The length property reflects numbered elements only; it does not include named array elements, which are treated as properties of the *Array* object. Although we ordinarily use *Array* class methods to change length indirectly, length can also be manipulated directly. Setting the length of an array changes the number of elements in the array. If we increase length, empty elements are added to the end of the array and assigned the value undefined. If we decrease length, existing elements are removed from the end of the array without warning; therefore *splice( )* or *pop( )* are the preferred methods of deleting elements from an array.

Example

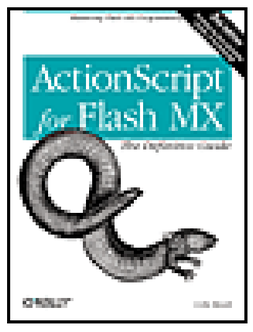
```
myList = new Array("one", "two", "three");
trace(myList.length);  // Displays: 3

// Loop through the array's elements
for (var i = 0; i < myList.length; i++) {
  trace(myList[i]);
}
```

See Also

Section 11.5

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir



ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Chapter 18. ActionScript Language Reference

Array.pop( ) Method Flash 5

remove the last element of an array

array.pop()

Returns

The value of the last element of *array*, which is also deleted.

Description

The *pop( )* method deletes the *last* element of an array, reduces the array's length by 1, and returns the value of the deleted element. Compare it with the *shift( )* method, which deletes the *first* element of an array.

Example

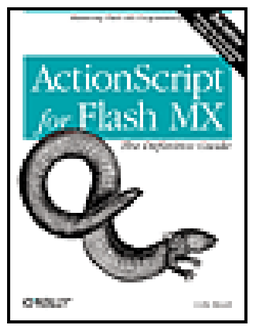
```
myList = new Array("one", "two", "three");
trace ("Now deleting " + myList.pop());  // myList is now: ["one", "two"]
```

See Also

Array.push( ), *Array.shift( )*, *Array.splice( )*; Section 11.8; Section 5.12.3

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir



Chapter 18. ActionScript Language Reference

Array.push( ) Method — Flash 5

add one or more elements to the end of an array

array.push(*value1, value2,...valuen*)

Arguments

value1, ...valuen

A list of one or more values to be added to the end of *array*.

Returns

The new length of *array*.

Description

The *push( )* method appends a list of values to the end of an array as new elements. Elements are added in the order provided. It differs from *concat( )* in that *push( )* modifies the original array, whereas *concat( )* creates a new array. It differs from *unshift( )* in that *push( )* adds elements at the end of an array, not the beginning.

Example

```
myList = new Array (5, 6);
myList.push(7);              // myList is now [5, 6, 7]
myList.push(10, 8, 9);       // myList is now [5, 6, 7, 10, 8, 9]
```

See Also

Array.concat( ), *Array.pop( )*, *Array.unshift( )*; "Adding Elements to an Array," in Chapter 11

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Chapter 18. ActionScript Language Reference

Array.reverse( ) Method Flash 5

reverse the order of elements in an array

array.reverse()

Description

The *reverse( )* method reverses the order of elements in an array, swapping the last element with the first element, the second-to-last element with the second element, and so on.

Example

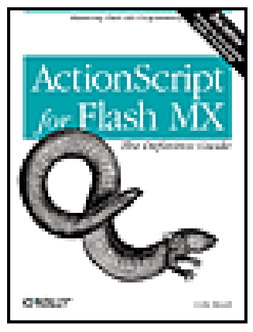
```
myList = new Array(3, 4, 5, 6, 7);
myList.reverse();                          // myList is now [7, 6, 5, 4, 3]
```

See Also

Array.sort( )

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir



ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Array.shift( ) Method Flash 5

remove the first element of an array

```
array.shift()
```

Returns

The value of the first element of *array*, which is also deleted.

Description

The *shift( )* method deletes the *first* element of an array and then moves all the remaining elements of the array up one position. The affected array's `length` is reduced by 1. Note that *shift( )* differs from the *pop( )* method, which deletes the *last* element of an array.

Example

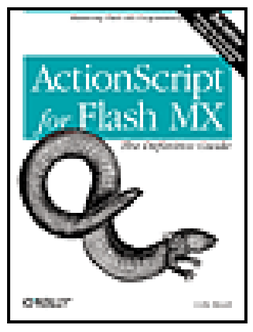
```
myList = new Array ("a", "b", "c");
myList.shift();                          // myList becomes ["b", "c"]
```

See Also

Array.pop( ), *Array.splice( )*, *Array.unshift( )*; "Removing Elements from an Array," in Chapter 11

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir



ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Array.slice( ) Method Flash 5

create a new array using a subset of elements from an existing array

```
array.slice(startIndex, endIndex)
```

Arguments

startIndex

A zero-relative integer specifying the first element of `array` to add to the new array. If negative, `startIndex` indicates an element number counting backward from the end of `array` (-1 is the last element, -2 is the second-to-last element, etc.).

endIndex

An integer specifying the element *after* the last element of `array` to add to the new array. If negative, `endIndex` counts backward from the end of `array` (-1 is the last element, -2 is the second-to-last element, etc.). If omitted, `endIndex` defaults to `array.length`.

Returns

A new array containing the elements of `array` from `startIndex` to `endIndex-1`.

Description

The *slice( )* method creates a new array by extracting a series of elements from an existing array. The original array is not modified. The new array is a subset of the elements of the original `array`, starting with `array[startIndex]` and ending with `array[endIndex - 1]`.

Example

```
myList = new Array("a", "b", "c", "d", "e");

// Set myOtherList to ["b", "c", "d"]
myOtherList = myList.slice(1, 4);

// Set anotherList to ["d", "e"]
anotherList = myList.slice(3);

// Set yetAnotherList to ["c", "d"]
yetAnotherList = myList.slice(-3, -1);
```

downloaded from: lib.ommolketab.ir

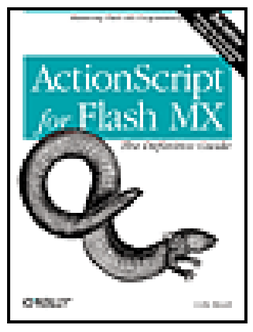
downloaded from: lib.ommolketab.ir

See Also

Array.splice( ); "The *slice( )* Method," in Chapter 11; "The *delete* Operator," in Chapter 5

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir



ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Chapter 18. ActionScript Language Reference

Array.sort( ) Method — Flash 5

sort the elements of an array

```
array.sort()
array.sort(compareFunction)
```

Arguments

compareFunction

An optional function that dictates how to sort `array`.

Description

When invoked without the optional `compareFunction` argument, *sort( )* temporarily converts the elements of `array` to strings and orders the elements according to the Unicode code points of those strings (approximately alphabetical order for Western European languages). Alphabetic comparisons and code points are described in Section 4.6.2.2 in Chapter 4.

When invoked with a `compareFunction` argument, *sort( )* reorders the elements of `array` according to the return value of `compareFunction`, which is a user-defined function that dictates how to sort any two values in the array. Your user-defined `compareFunction` should be designed to accept two array elements as arguments. It should return a negative number if the first element should come before the second element; it should return a positive number if the first element should come after the second element; and it should return a 0 if the elements should not be reordered. If additional elements are added to the array after it has been sorted, they are *not* added in sorted order. You must resort the array to reorder any newly added elements. Note that numbers are sorted according to their Unicode code points by default (which are the same code points as Latin 1 and ASCII). Chapter 11 explains how to sort numbers by their numeric values. To ensure case-insensitive alphabetic sorting for Latin 1 letters, use *String.toUpperCase( )* to convert strings to uppercase before calling *sort( )* or inside the `compareFunction`, as shown in Example 11-6.

To sort a group of objects by a given property's value, use the *sortOn( )* method.

Bugs

In Flash Player 5, *Array.sort( )* has several sorting errors, described at: http://moock.org/asdg/technotes/arraySortBugs. These are fixed in Flash Player 6.

Example

The following example sorts an array of movie clips according to their horizontal location on screen. This is useful to

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

create a fade effect that travels from left to right across the screen and fades the movie clips based on their position, even if that position changes dynamically.

```
var clips = [clip1, clip2, clip3, clip4];

function compareXposition (element1, element2) {
  if (element1._x < element2._x) {
    return -1;
  } else if (element1._x > element2._x) {
    return 1;
  } else {
    return 0;  // The clips have the same x position
  }
}

clips.sort(compareXposition);
```

See Also

Array.reverse( ), *Array.sortOn( )*; "The *sort( )* and *onsort( )* Methods," in Chapter 11

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Array.sortOn( ) Method — Flash 6

sort an array of objects by the specified property

array.sortOn(*key*)

Arguments

key

A mandatory string specifying which property should be used to sort the objects in *array*.

Description

The *sortOn( )* method is used to sort an array of objects according to the value of a specified property. The elements of *array* are expected to be objects. Each object is expected to have a property, specified by the string *key*, that will be used to sort the object. If *key* is not provided, *sortOn( )* silently fails and no sort is performed.

The following code populates a three-element array with object literals that define `fruit` and `meat` properties:

```
var shoppingList = new Array();
shoppingList[0] = {fruit:"mango", meat:"beef"};
shoppingList[1] = {fruit:"orange", meat:"pork"};
shoppingList[2] = {fruit:"apple", meat:"chicken"};
```

To sort the `shoppingList` array by the `fruit` property, we use:

```
shoppingList.sortOn("fruit");
// shoppingList now contains:
// ELEMENT 0: {fruit:"apple", meat:"chicken"}
// ELEMENT 1: {fruit:"mango", meat:"beef"}
// ELEMENT 2: {fruit:"orange", meat:"pork"};
```

More typically, the objects in the array will be instances of some class. For example:

```
// Create a simple Contact class with firstName and lastName properties
function Contact (firstName, lastName) {
  this.firstName = firstName;
  this.lastName = lastName;
}

// Override toString( ) for our class so it returns a more legible value
Contact.prototype.toString = function () {
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

```
  return this.firstName + " " + this.lastName;
}

// Populate a contacts array with Contact instances
var contacts = new Array();
contacts[0] = new Contact("Colin", "Moock");
contacts[1] = new Contact("Bruce", "Epstein");
contacts[2] = new Contact("Derek", "Clayton");

// Sort by lastName...
contacts.sortOn("lastName");
trace(contacts);  // Displays: Derek Clayton,Bruce Epstein,Colin Moock
// Sort by firstName...
contacts.sortOn("firstName");
trace(contacts);  // Displays: Bruce Epstein,Colin Moock,Derek Clayton
```

Note that *sortOn( )* uses the standard compare function to sort values (as if *sort( )* had been called with no arguments). The *sortOn( )* method does not offer any custom compare function capabilities. For sorting routines that require a custom compare function, use the *sort( )* method instead. To ensure case-insensitive alphabetic sorting for Latin 1 letters, use *String.toUpperCase( )* to convert property values to uppercase before calling *sortOn( )*.

When an element does not contain an object with a `key` property, that element is converted to a string and then ordered according to the default sort rules.

Bugs

In Flash Player 6, using *sortOn( )* on an array that contains both strings and objects can crash the Player and the browser in which it is running. For example, the following code crashes Flash Player 6:

```
var shoppingList = new Array();
shoppingList[0] = {fruit:"mango", meat:"beef"};
shoppingList[1] = "hi";
shoppingList[2] = {fruit:"apple", meat:"chicken"};

shoppingList.sortOn("fruit");
```

Therefore, it is advisable to user a wrapper function that first checks whether all elements are objects before calling *sortOn( )*, such as:

```
Array.prototype.safeSortOn = function (key) {
  for (i=0; i < this.length; i++) {
    if (typeof this[i] != "object") {
      trace ("Element " + i + " is not an object.");
      return;
    }
  }
  // If it passes the test, then call sortOn()
  this.sortOn(key);
}

// Usage:
// This won't crash Flash Player 6.
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

```
shoppingList.safeSortOn("fruit");
```

The sort order is indeterminate when multiple elements have the identical value for the *key* property. In this case, *sortOn( )* may produce inconsistent results between multiple calls. For example:

```
function Contact (firstName, lastName) {
  this.firstName = firstName;
  this.lastName = lastName;
}

Contact.prototype.toString = function () {
  return this.firstName + " " + this.lastName;
}

// Populate a contacts array with Contact instances that
// share identical lastName property values.
var contacts = new Array();
contacts[0] = new Contact("Makiko", "Pitaru");
contacts[1] = new Contact("Amit",   "Pitaru");
contacts[2] = new Contact("Richard", "Clayton");
contacts[3] = new Contact("Derek",   "Clayton");

// Sort by lastName...
contacts.sortOn("lastName");
trace(contacts);  // Displays:
                  // Richard Clayton,Derek Clayton,Makiko Pitaru,Amit Pitaru

// Sort AGAIN by lastName...
contacts.sortOn("lastName");
trace(contacts);  // Displays DIFFERENT RESULTS:
                  // Richard Clayton,Derek Clayton,Amit Pitaru,Makiko Pitaru
```

See Also

Array.sort( )

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Chapter 18. ActionScript Language Reference

Array.splice( ) Method — Flash 5

remove elements from, and/or add elements to, an array

```
array.splice(startIndex)
array.splice(startIndex,
deleteCount)
array.splice(startIndex, deleteCount, value1,...valuen)
```

Arguments

startIndex

The zero-relative element index at which to start element deletion and optional insertion of new elements. If negative, `startIndex` specifies an element counting back from the end of `array` (-1 is the last element, -2 is the second-to-last element, etc.).

deleteCount

An optional, nonnegative integer representing the number of elements to remove from `array`, including the element at `startIndex`. If `deleteCount` is 0, no elements are deleted. If `deleteCount` is omitted, all elements from `startIndex` to the end of the array are removed.

value1, ...valuen

An optional list of one or more values to be added to `array` at index `startIndex` after the specified elements have been deleted. If this list is omitted, no elements are added.

Returns

A new array containing the *deleted* elements (the original `array` is modified separately to reflect the requested changes).

Description

The *splice( )* method removes the elements from `array[startIndex]` to `array[startIndex + deleteCount - 1]` and then, optionally, inserts new elements starting at `startIndex`. The *splice( )* method does not leaves gaps in `array`; it compacts deleted elements and moves remaining elements as needed to ensure the contiguity of elements in the array.

Example

```
myList = new Array (1, 2, 3, 4, 5);
// Deletes the second and third elements from the list
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

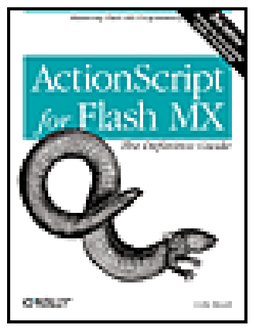
```
// and inserts the elements "x", "y", and "z" in their place.
// This changes myList to [1, "x", "y", "z", 4, 5].
myList.splice(1, 2, "x", "y", "z");
```

See Also

Array.slice( ); "The *splice( )* Method," in Chapter 11

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir



Chapter 18. ActionScript Language Reference

Array.toString( ) Method — Flash 5

convert an array to a string of comma-separated element values

array.toString()

Returns

A comma-separated list of *array*'s elements, converted to strings.

Description

The *toString( )* method creates a string representation of *array*. The string returned by *toString( )* is a list of array elements converted to strings and separated by commas (the same as is returned by the *join( )* method when *join( )* is invoked without parameters). An array's *toString( )* method is invoked automatically whenever the array is used in a string context. Therefore, it is rarely necessary to invoke *toString( )* manually. Normally, when we want a precise string representation of an array, we use the *join( )* method, which offers more control over the string we're creating.

Example

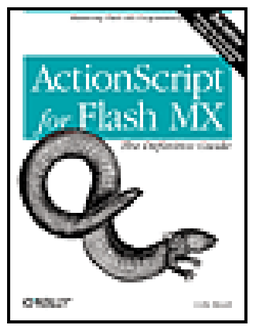
```
myList = new Array("a", "b", "c");              // Create an array
trace(myList.toString());                       // Displays: "a,b,c"
myList = new Array([1, 2, 3], "a", "b", "c");   // Create a nested array
trace(myList.toString());                       // Displays: "1,2,3,a,b,c"
```

See Also

Array.join( )

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir



Chapter 18. ActionScript Language Reference

Array.unshift( ) Method — Flash 5

add one or more elements to the beginning of an array

```
array.unshift(value1, value2,...valuen)
```

Arguments

value1, ...valuen

A list of one or more element values to be added to the beginning of `array`.

Returns

The new `length` of `array`.

Description

The *unshift( )* method adds a series of elements to the beginning of an array. Elements are added in the order specified. To add elements at the end of an array, use *push( )*.

Example

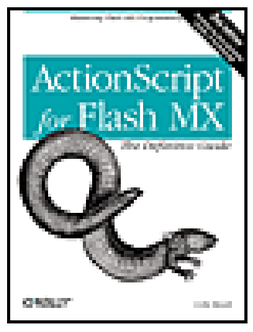
```
myList = new Array (5, 6);
myList.unshift(4);          // myList becomes [4, 5, 6]
myList.unshift(7, 1);       // myList becomes [7, 1, 4, 5, 6]
```

See Also

Array.push( ), *Array.shift( )*; "Adding Elements to an Array," in Chapter 11

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir



ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Boolean( ) Global Function — Flash 5

convert a value to the boolean datatype

`Boolean(`*`value`*`)`

Arguments

value

An expression containing the value to be converted to a Boolean.

Returns

The result of converting *`value`* to a primitive Boolean (either `true` or `false`).

Description

The *Boolean( )* global function converts its argument to a primitive Boolean value and returns that converted value. The results of converting various datatypes to a primitive Boolean are described in Table 3-3 in Chapter 3. It's normally not necessary to use the *Boolean( )* function; ActionScript automatically converts values to the *boolean* type when appropriate.

Usage

Be sure not to confuse the global *Boolean( )* function with the *Boolean* class constructor. The *Boolean( )* function converts its argument to the *boolean* datatype, whereas the *Boolean* class constructor creates a new *Boolean* object. Note that in ECMA-262, all nonempty strings convert to `true`. In Flash, only strings that can be converted to a valid nonzero number convert to `true`. Therefore, even the string "true" converts to the Boolean `false`.

Example

```
var x = 1;
if (Boolean(x)) {
  trace("x is true");
}
```

See Also

The *Boolean* class; "Datatype Conversion," in Chapter 3

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Chapter 18. ActionScript Language Reference

Boolean Class — Flash 5

wrapper class for primitive Boolean data

Constructor

`new Boolean(`*`value`*`)`

Arguments

value

An expression to be resolved and, if necessary, converted to a Boolean value, then wrapped in a *boolean* object. If the *`value`* argument is omitted, the stored value will be the Boolean `false`.

Methods

toString( )

Convert the value of a *Boolean* object to a string.

valueOf( )

Retrieve the primitive value of a *Boolean* object.

Description

The *Boolean* class creates a *Boolean* object, which contains a primitive Boolean value in an internal property. *Boolean* objects are used purely for the sake of manipulating and examining primitive Boolean values using methods of the *Boolean* class. Hence, a *Boolean* object is known as a *wrapper object* because it simply packages a primitive Boolean value, giving it some object-like methods. Compare the *Boolean* class with the *String* and *Number* classes, which similarly wrap *string* and *number* primitive values, but with more useful results.

For the most part, *Boolean* objects are used internally. They are created automatically by the interpreter whenever a method is invoked on a primitive Boolean value and are deleted automatically after each use. We can create *Boolean* objects ourselves using the *Boolean* constructor, but there is seldom reason to do so.

Usage

Note that in practice it is much more common to use the *Boolean( )* global function as a datatype-conversion

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

tool than it is to use the *Boolean* class.

See Also

Boolean( ) global function; "The Boolean Type," in Chapter 4

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Boolean.toString( ) Method Flash 5

the value of the Boolean object converted to a string

booleanObject.toString()

Returns

The string "true" if the primitive value of *booleanObject* is `true`; "false" if the primitive value of *booleanObject* is `false`. The value of the Boolean object is specified when the object is constructed, and it is stored internally thereafter. Although the internal value of a Boolean object is obtainable via *valueOf( )*, we can use *toString( )* to convert it to its string equivalent.

Description

The *toString( )* method retrieves the primitive value of a Boolean object, converts that value to a string, and returns the resulting string.

Example

```
x = new Boolean(true);
trace(x.toString());  // Displays: "true"
```

See Also

Object.toString( )

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Chapter 18. ActionScript Language Reference

Boolean.valueOf( ) Method Flash 5

the primitive value of the Boolean object

booleanObject.valueOf( )

Returns

The Boolean value `true` if the primitive value of *`booleanObject`* is `true`; `false` if the primitive value of *`booleanObject`* is `false`. The value of the Boolean object is specified when the object is constructed, and it is stored internally thereafter.

Description

The *valueOf( )* method returns the primitive Boolean datum associated with a Boolean object. The *valueOf( )* method returns the *`value`* argument with which the constructor was called, converted to its primitive Boolean equivalent.

Example

```
x = new Boolean(0);
trace(x.valueOf());  // Displays: false
```

See Also

Object.valueOf( )

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Chapter 18. ActionScript Language Reference

Button Class — Flash 6

control over buttons in a movie.

Constructor

None. *Button* objects must be created manually in the authoring tool. Use movie clips to simulate runtime-generated buttons.

Properties

_alpha

Opacity percentage: 0 is transparent; 100 is opaque.

enabled

Boolean; allows or disallows button interaction.

_focusrect

Boolean; enables or disables automatic highlighting of buttons activated via the keyboard.

_height

Height of the button, in pixels.

_name

The instance name of the *Button* object.

_parent

A reference to the movie clip in which the button resides.

_rotation

Clockwise rotation, in degrees, of the button.

tabEnabled

Boolean; includes or excludes the button from the current tab order.

tabIndex

Specifies the button's index in the custom tab order.

_target

The target path of the button, in slash syntax.

trackAsMenu

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Modifies the button's *onRelease( )* handler requirements, enabling menu-style behavior.

_url

The network address of the movie clip that contains the button.

useHandCursor

Boolean; dictates whether a hand icon is displayed when the mouse is over the button.

_visible

Boolean; controls whether the button is shown or hidden.

_width

Width of the button, in pixels.

_x

Horizontal location of the button's registration point, in pixels.

_xmouse

Horizontal location of mouse pointer, relative to the button's registration point, in pixels.

_xscale

Width of the button, as a percentage of its original width.

_y

Vertical location of the button's registration point, in pixels.

_ymouse

Vertical location of mouse pointer, relative to the button's registration point, in pixels.

_yscale

Height of the button, as a percentage of its original height.

Methods

getDepth( )

Returns the position of the button in the content stack.

Event Handlers

on (keyPress)

Occurs when the specified key is pressed.

onDragOut( )

Occurs when the mouse is clicked while the pointer is over a button and then the pointer moves off the Flash button without the mouse button being released.

onDragOver( )

Occurs when the mouse is clicked while the pointer is over a button and then the pointer moves off and back onto the Flash button without the mouse button being released.

onKillFocus( )

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Occurs when the button loses focus.

onPress( )

Occurs when the mouse is clicked while the pointer is over a Flash button.

onRelease( )

Occurs when the mouse is pressed and then released while the pointer is over a Flash button.

onReleaseOutside( )

Occurs when the mouse is clicked over the Flash button and then released after rolling off the button.

onRollOut( )

Occurs when the pointer moves off the Flash button (while the mouse is not depressed).

onRollOver( )

Occurs when the pointer moves over the Flash button (while the mouse is not depressed).

onSetFocus( )

Occurs when the Flash button gains focus.

Description

The *Button* class provides programmatic control over the buttons in a movie. Each *Button* object represents a single Button symbol instance created manually at authoring time. Buttons cannot be created or destroyed via ActionScript, even though the *Button* class constructor technically exists for the sake of containing the class's methods and properties. That is, calling *new Button( )* does not create a visual button instance for use in a movie. However, movie clips—which have all the capabilities of buttons—can be both created and destroyed via ActionScript at runtime. To remove a button instance, place it in movie clip, and remove the clip using *MovieClip.removeMovieClip( )*.

Prior to Flash 6, buttons were not accessible to ActionScript. As of Flash 6, the *Button* class lets us manipulate buttons much as we do movie clips, by setting position, size, rotation, etc. In fact, many button properties are borrowed directly from the *MovieClip* class (for example, `_x`, `_y`, `_width`, `_height`, and `_rotation`). For the sake of backward compatibility, button properties corresponding to movie clip properties that existed prior to Flash 6 are prefixed with an underscore (_). To conform more closely with other ECMA-262 languages, properties that are completely new in Flash 6 do not use the underscore convention.

The *Button* class's event handlers provide tools for creating navigation, forms, games, and other interface elements. Each type of event is described in detail later in this entry. To respond to events, a button object can use either of two different types of event handlers — an event handler property:

```
go_btn.onPress = function () {
  // statements
};
```

or the idiosyncratic Flash 5 notation attached directly to the button in the authoring tool:

```
on (press) {
  // statements
}
```

However, one of the button events, *keyPress*, supports the Flash 5 *on(keyPress)* form only. See *Button keyPress* for details.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

For general information on creating event handlers, see Chapter 10. Note that as of Flash 6, all button event handlers are also supported by the *MovieClip* class.

Button events are sent only to the button with which the mouse is interacting. If multiple buttons overlap, the topmost button receives all events; no other buttons can respond, even if the topmost button has no handlers defined.

In the following entries for *Button* property, method, and event descriptions:

- The *hit area* refers to the physical region of the button that must be under the mouse pointer in order for the Flash button to be activated. (A Flash button's hit area is defined graphically when you create the button in the Flash authoring tool.)
- The *Up*, *Over*, and *Down* states correspond to the button frames of the same name in the Flash authoring tool. These are displayed automatically when the mouse is not over the Flash button's hit area (*Up*), the mouse is over the Flash button's hit area (*Over*), or the mouse is depressed over the Flash button's hit area (*Down*).

To enable Actions panel code-hinting for button objects, use an instance name that includes the suffix "`_btn`", as in: `submit_btn`.

Usage

Using *Button* objects, even novice developers can add interactivity with minimal effort. However, for advanced applications, movie clips are often more appropriate than buttons. Movie clips offer all the capabilities of buttons, but with added features such as a dynamic hit area, changeable button states (*Up*, *Over*, and *Down*), and the full suite of *MovieClip* methods, events, and properties.

See Also

The *MovieClip* class; Chapter 10

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Chapter 18. ActionScript Language Reference

Button._alpha Property Flash 6

opacity percentage: 0 is transparent; 100 is opaque *read/write*

theButton._alpha

Description

The floating-point _alpha property specifies the opacity (or, conversely, the transparency) of *theButton* as a percentage—0 is completely transparent, whereas 100 is completely opaque. All states (*Up*, *Over*, and *Down*) of *theButton* are affected.

Example

The following code makes go_btn 50% transparent.

```
go_btn._alpha = 50;
```

See Also

Button._visible, *MovieClip*._alpha

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Chapter 18. ActionScript Language Reference

Button.enabled Property Flash 6

Boolean; allows or disallows button interaction ***read/write***

theButton.enabled

Description

The Boolean `enabled` property turns a button on (`true`) and off (`false`). It defaults to `true`. When `enabled` is `false`, *`theButton`* does not receive any user input; the hand cursor is not shown, no events are triggered, and only the *Up* state is displayed. However, a disabled button can still receive keyboard focus with the Tab key. To disable tab focus, set *`theButton`*`.tabEnabled` to `false`.

Usage

The Tab key does not work reliably in Test Movie mode unless Control Disable Keyboard Shortcuts is enabled. Be sure to test keyboard behavior in the Standalone Player, if applicable, and in the Flash Player for all targeted web browsers.

Example

Buttons can be disabled in order to control a user's interaction with an interface. For example, a form's Submit button can be disabled until all of the form's input text fields have been filled out by the user:

```
// Disable the Submit button at the outset
submit_btn.enabled = false;
submit_btn.tabEnabled = false;

// Add code that reenables the button when all fields have been filled out..
// For example:
mandatoryField.onChanged = function () {
  if (this.text != "") {
    submit_btn.enabled = true;
    submit_btn.tabEnabled = true;
  }
}
```

See Also

Button.tabEnabled, *Button*.useHandCursor

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Chapter 18. ActionScript Language Reference

Button._focusrect Property — Flash 6

Boolean; enables or disables automatic highlighting of buttons activated via the keyboard — ***read/write***

theButton`._focusrect`

Description

When the mouse pointer hovers over a Flash button, the content of the button's *Over* state is displayed. Buttons can also gain keyboard focus when the user presses the Tab key. When a button has keyboard focus, Flash displays the button's *Over* state and also places a yellow rectangle around the button, which is not always aesthetically desirable. You can turn off the yellow rectangle for an individual button by setting its `_focusrect` property to `false`:

```
go_btn._focusrect = false;
```

When `_focusrect` is set to `false`, Flash displays only the *Over* state of keyboard-focused buttons. When `_focusrect` is set to `true`, Flash displays the *Over* state and the yellow rectangle. When it is `null` (the default), the value of the global `_focusrect` property is used. To disable the yellow focus rectangle for all buttons in a movie, set the global `_focusrect` property to `false`. Note that the value of the global `_focusrect` property is not reflected by the `_focusrect` property of individual button instances. Furthermore, setting an individual button's `_focusrect` property overrides the global setting for that button.

Usage

The Tab key does not work reliably in Test Movie mode unless Control Disable Keyboard Shortcuts is enabled. Be sure to test keyboard behavior in the Standalone Player, if applicable, and in the Flash Player for all targeted web browsers.

See Also

`_focusrect`, *MovieClip*`._focusrect`

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Chapter 18. ActionScript Language Reference

Button.getDepth( ) Method Flash 6

returns the position of the button in the content stack

```
theButton.getDepth()
```

Returns

An integer depth level.

Description

Buttons, movie clips, and text fields are all placed in the Player in an order that governs how objects stack visually. An internal integer depth property indicates each object's order in the stack. The *getDepth( )* method tells us `theButton`'s integer depth position. Each depth level can have only one occupant, whether it is a text field, button, or movie clip. Objects on higher depths appear in front of those with lower depths.

Because all buttons are created at authoring time, *Button.getDepth( )* always returns a depth in the range of -16383 to 0 (which is reserved for authored content). Unlike the *MovieClip* class, *Button* has no *swapDepths( )* method, so each button's depth is fixed once it is assigned.

For much more information about depths, see Section 13.4 in Chapter 13.

Example

Check the depth of `cancel_btn`, placed on the lowest layer of the main timeline:

```
trace(cancel_btn.getDepth());  // Displays: -16383
```

See Also

MovieClip.getDepth( ), *MovieClip.swapDepths( )*, *TextField.getDepth( )*; "Movie and Instance Stacking Order," in Chapter 13

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Chapter 18. ActionScript Language Reference

Button._height Property — Flash 6

height of the button, in pixels — ***read/write***

theButton._height

Description

The floating-point _height property is a nonnegative number specifying the height of *theButton*'s current frame (*Up*, *Over*, *Down*), in pixels. It can be both retrieved and set (the latter action changes the button's size on screen). In all other respects, it behaves identically to *MovieClip*._height; see that property entry under the *MovieClip* class for details.

See Also

Button._width, *MovieClip*._height

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Button keyPress Event Handler — Flash 4

occurs when the specified key is pressed

```
on (keyPress key) {
  statements
}
```

Description

The *keyPress* event is triggered by the pressing of a specified key. It uses the Flash 5-style *on(`eventName`)* syntax and is not available in the more modern callback-function form. In Flash 6, *Key.onKeyUp( )* and *Key.onKeyDown( )* are the preferred keyboard-handling tools, because they do not require an on-stage object and support the detection of multiple simultaneous key presses.

To trap key events, we specify the key that triggers the event, using the syntax listed at the beginning of this entry, where *`key`* is a string representing the key associated with the event. The string can be either the character corresponding to the key (such as `"s"` or `"S"` , because it is case-insensitive) or a keyword representing the key in the format `"<Keyword>"` . Only one key can be specified with each handler. To capture multiple keys using *keyPress* , we must create multiple *keyPress* event handlers. For example:

```
// Detects the "a" key
on (keyPress "a") {
  trace("The 'a' key was pressed");
}

// Detects the Enter key
on (keyPress "<Enter>") {
  trace("The Enter key was pressed");
}

// Detects the down arrow key
on (keyPress "<Down>") {
  trace("The down arrow key was pressed");
}
```

The legal values of *`Keyword`* are as follows (note that the function keys, F1 through F12, are not supported by *keyPress* but are detectable using the *Key* object):

```
<Backspace>
<Delete>
<Down>
<End>
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

```
<Enter>
<Home>
<Insert>
<Left>
<PgDn>
<PgUp>
<Right>
<Space>
<Tab>
<Up>
```

In Flash 4, *keyPress* was the only means of interacting with the keyboard.

Usage

Unlike all other keyboard event-handling tools in ActionScript, the *keyPress* Button event intercepts the default keyboard behavior in the Player. For example, if a button traps the left arrow key via *keyPress* , the left arrow will not move the I-beam cursor in a text field.

```
on ( keyPress "<Left>" ) {
  // We're just blocking the <Left> key, but we must provide some code here
  // or this function will get optimized out.
  1 + 1;
}
```

See Also

Key.onKeyDown( ) , Key.onKeyUp( )

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Chapter 18. ActionScript Language Reference

Button._name Property — Flash 6

the instance name of the Button object — ***read/write***

theButton._name

Description

The string _name property indicates *theButton*'s instance name, which is specified via the Property inspector at authoring time. Though rarely (if ever) necessary, setting _name reassigns the instance name for *theButton*; thereafter, references to the old instance name yield undefined.

More commonly, the _name property is read, not written. For example, it can be used to identify a specific button when one of many is clicked. The following function performs different actions based on the calling button:

```
function doAction () {
  if (this._name =  = "ok_btn") {
    doConfirm();
  } else if (this._name =  = "cancel_btn") {
    doCancel();
  }
}

// Assign button handlers
ok_btn.onRelease = doAction;
cancel_btn.onRelease = doAction;
```

Or, if we're looking for a single button in a group of unknown buttons, we can use a *for-in* loop to hunt for the button's instance name:

```
// Access all properties of someClip
for (var p in someClip) {
  // If it's a Button object...
  if (someClip[p] instanceof Button) {
    // ...show its name:
    trace("Found " + someClip[p]._name);

    // Look for edit_btn
    if (someClip[p]._name =  = "edit_btn") {
      // Disable the edit button.
      someClip[p].enabled = false;
    }
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

```
    }
}
```

See Also

Button.onRelease( ), "The *for-in* Loop," in Chapter 8

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Chapter 18. ActionScript Language Reference

Button.onDragOut( ) Event Handler — Flash 6; on (dragOut) supported since Flash 2

occurs when the depressed mouse is rolled off a Flash button

```
theButton.onDragOut()

on (dragOut) {
  statements
}
```

Description

The *onDragOut( )* event handler executes when the primary mouse button is depressed while the mouse pointer is in `theButton`'s hit area, and then, while the mouse button is still depressed, the pointer is moved out of the hit area. It is similar to *onRollOut( )*, except that it is generated only if the mouse button is down when the pointer leaves a button's hit area. The *onDragOut( )* event is followed by either the *onReleaseOutside( )* event (if the user releases the mouse button) or the *onDragOver( )* event (if the user moves the pointer back into the button's hit area without releasing the mouse button).

Example

```
cancel_btn.onDragOut = function () {
  // Handle event...
  trace("dragOut detected");
};
```

See Also

Button.onDragOver( ), *Button.onRollOut( )*, *MovieClip.onDragOut( )*; Chapter 10

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

occurs when the depressed mouse rolls off and back onto a Flash button

```
theButton.onDragOver()

on (dragOver) {
  statements
}
```

Description

The *onDragOver( )* event handler is a seldom-seen woodland creature. It is conjured up when the following sequence is performed:

1. The mouse pointer moves into the hit area of a button (*onRollOver( )* event occurs).
2. The primary mouse button is pressed and held (*onPress( )* event occurs).
3. The mouse pointer moves out of the button's hit area (*onDragOut( )* event occurs).
4. The mouse pointer moves back into the button's hit area (*onDragOver( )* event occurs).

Thus, the *dragOver* event indicates that the user has moved the mouse pointer out of and back into the hit area, all the while holding down the mouse button. Note that *dragOver*, instead of the *rollOver* event, is generated if the mouse button is still down when the pointer reenters the button's hit area.

Example

```
cancel_btn.onDragOver = function () {
  // Handle event...
  trace("dragOver detected");
};
```

See Also

Button.onDragOut( ), *Button.onRollOver( )*, *MovieClip.onDragOver( )*; Chapter 10

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Chapter 18. ActionScript Language Reference

Button.onKillFocus( ) Event Handler Flash 6

callback invoked when the button loses focus

```
theButton.onKillFocus(newFocus)
```

Arguments

newFocus

The *TextField*, *MovieClip*, or *Button* object that now has focus, or `null` if no object has received focus.

Description

The *onKillFocus( )* event handler is triggered when `theButton` loses focus. A button is said to "have focus" when:

- The user navigates to it using the Tab key.
- *Selection.setFocus( )* assigns it focus programmatically.

When `_focusrect` is `true` or `null` (`null` is the default) a yellow rectangle appears around the focused button. A button loses focus when:

- The user navigates to a different button, text field, or movie clip using the Tab key.
- *Selection.setFocus( )* assigns focus to another object programmatically. (The newly focused object is accessible within *onKillFocus( )* via `newFocus`.)
- The user moves the mouse.

In the first two cases, *onKillFocus( )* is executed. In the last case, due to a bug in Flash Player 6, *onKillFocus( )* is not executed, but it should be. When a button's `tabEnabled` property is `false`, the button cannot be focused by the user but can be focused programmatically with *Selection.setFocus( )*. To capture all focus events centrally, rather than for a single instance, use *Selection.onSetFocus( )*.

Example

The following example adds callback handlers, for *onSetFocus( )* and *onKillFocus( )*, to `cancel_btn`. It also creates a text field that displays the focus status. Finally, it focuses `cancel_btn`. To try the code, create two buttons, one called "cancel_btn". Then run the movie in the Standalone Player and press the Tab key (*without*

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

touching the mouse!). If you test in a browser, you must first click the movie before pressing Tab.

```
this.createTextField("status_txt", 1, 0, 0, 300, 20);
status_txt.border = true;

cancel_btn.onKillFocus = function () {
  // Handle event...
  status_txt.text = ("killfocus detected for cancel_btn");
};

cancel_btn.onSetFocus = function () {
  // Handle event...
  status_txt.text = ("setfocus detected for cancel_btn");
};

Selection.setFocus(cancel_btn);
```

See Also

Button.onSetFocus( ), *Selection.onSetFocus( )*, *Selection.setFocus( )*, *MovieClip.onKillFocus( )*, *TextField.onKillFocus( )*; Chapter 10

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Chapter 18. ActionScript Language Reference

Button.onPress( ) Event Handler — Flash 6; on (press) supported since Flash 2

occurs when the mouse button is clicked while the pointer is over a Flash button

```
theButton.onPress()

on (press) {
  statements
}
```

Description

The *onPress( )* event handler executes when the primary mouse button is depressed while the mouse pointer is in the button's hit area. Other mouse buttons (right, middle, scroll, etc.) are not detectable. Note that a "mouseclick" is a two-step process: the mouse button is depressed (*press*) and then released (*release*). Button press events are appropriate for radio buttons or weapons firing in a game, but use release events to allow the user to change his mind before releasing the mouse.

Example

```
cancel_btn.onPress = function () {
  // Handle event...
  trace("press detected");
};
```

See Also

Button.onRelease( ), *MovieClip.onPress( )*; Chapter 10

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Chapter 18. ActionScript Language Reference

Button.onRelease( ) Event Handler — Flash 6; on (release) supported since Flash 2

occurs when the mouse is pressed and then released while the pointer is over a Flash button

```
theButton.onRelease()

on (release) {
  statements
}
```

Description

The *onRelease( )* event handler is executed when the following sequence is detected:

1. The mouse pointer is in the hit area of `theButton`.
2. The primary mouse button is pressed while the mouse pointer is still in the hit area of `theButton` (at which point an *onPress( )* event occurs).
3. The primary mouse button is released while the mouse pointer is still in the hit area of `theButton` (at which point an *onRelease( )* event occurs).

By using the release event instead of the press event, you give users a chance to move the pointer off of a button even after the mouse has been pressed, thus allowing them to retract their action. The *onRelease( )* handler is the most common button handler in Flash. It is used to respond to the classic button click. Note that the `trackAsMenu` property can alter the behavior of *onRelease( )*, as described under `Button.trackAsMenu`.

Example

The following code shows two *onRelease( )* handlers. When `ok_btn` is clicked, the playhead of the button's parent movie clip is sent to the frame labeled `begin`. When `cancel_btn` is clicked, the playhead is sent to the frame labeled `quit`.

```
ok_btn.onRelease = function () {
  this._parent.gotoAndStop("begin");
}

cancel_btn.onRelease = function () {
  this._parent.gotoAndStop("quit");
}
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

See Also

Button.onPress( ), *Button*`.trackAsMenu`, *MovieClip.onRelease( )*; Chapter 10

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Chapter 18. ActionScript Language Reference

Button.onReleaseOutside( ) Event Handler

Flash 6; on(releaseOutside) supported since Flash 2

---

occurs when the mouse is clicked over the button, then released after rolling off the button

```
theButton.onReleaseOutside()

on (releaseOutside) {
  statements
}
```

Description

The *onReleaseOutside( )* event typically indicates that the user changed his mind after clicking on a button, moving the pointer off the button before releasing the mouse button. The *onReleaseOutside( )* event handler is executed when the following sequence is detected:

1. The mouse pointer is in the hit area of `theButton`.
2. The primary mouse button is pressed and held (the *onPress( )* event occurs).
3. The mouse pointer moves out of `theButton`'s hit area (the *onDragOut( )* event occurs).
4. The primary mouse button is released while the pointer is no longer in the hit area of `theButton`.

You will rarely bother to detect *onReleaseOutside( )* events, as they usually indicate that the user intended not to perform any action.

Example

```
cancel_btn.onReleaseOutside = function () {
  trace("releaseOutside detected");
}
```

See Also

Button.onRelease( ), *MovieClip.onReleaseOutside( )*; Chapter 10

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Chapter 18. ActionScript Language Reference

Button.onRollOut( ) Event Handler — Flash 6; on(rollOut) supported since Flash 2

occurs when the pointer moves off the button (while the mouse is not depressed)

```
theButton.onRollOut()

on (rollOut) {
  statements}
```

Description

The *onRollOut( )* event handler is *onRollOver( )*'s counterpart; it executes when the mouse pointer is moved out of the hit area of a button while the mouse button is not depressed. As with *onRollOver( )*, *onRollOut( )* normally is not used to create button highlight states, because visual button changes are created directly in the authoring tool. You should use the provided *Up*, *Over*, and *Down* frames in the authoring tool to create highlight states for buttons.

Example

```
cancel_btn.onRollOut = function () {
  trace("rollOut detected");
};
```

See Also

Button.onDragOut( ), *Button.onRollOver( )*, *MovieClip.onRollOut( )*; Chapter 10

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Chapter 18. ActionScript Language Reference

Button.onRollOver( ) Event Handler — Flash 6; on(rollOver) supported since Flash 2

occurs when the pointer moves over the button (if mouse is not depressed) or when the button gains keyboard focus

```
theButton.onRollOver()

on (rollOver) {
  statements
}
```

Description

The *onRollOver( )* event handler executes when the mouse pointer moves into the hit area of a button without the mouse button depressed. It also executes when `theButton` gains keyboard focus, as described under *Button.onSetFocus( )*. The *rollOver* event is not required to depict visual states for a button, because visual button changes are created directly in the authoring tool, not with scripting. You should use the provided *Up*, *Over*, and *Down* frames in the authoring tool to create highlight states for buttons.

The *rollOver* event provides a handy means of retrieving a text field selection. For more details, see *TextField.replaceSel( )* and the *Selection* object.

Example

```
cancel_btn.onRollOver = function () {
  trace("rollOver detected");
};
```

See Also

Button.onDragOut( ), *Button.onRollOut( )*, *Button.onSetFocus( )*, *MovieClip.onRollOver( )*; Chapter 10

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Chapter 18. ActionScript Language Reference

Button.onSetFocus( ) Event Handler Flash 6

callback invoked when the button gains focus

```
theButton.onSetFocus(oldFocus)
```

Arguments

oldFocus

The *TextField*, *MovieClip*, or *Button* object that previously had focus, or `null` if no object was previously focused.

Description

The *onSetFocus( )* event handler is executed when `theButton` gains focus. A button gains focus when:

- The user navigates to it with the Tab key.
- *Selection.setFocus( )* assigns focus programmatically.

When a button's `tabEnabled` property is `false`, the button cannot be focused by the user but can be focused programmatically with *Selection.setFocus( )*. The following code registers a callback function that responds to *onSetFocus( )*:

```
// Assign the onSetFocus() callback using a function literal
next_btn.onSetFocus = function (oldFocus) {
  trace(this + " now has focus. Old focus was " + oldFocus);
};
```

Notice that from the body of the callback, we can refer to `next_btn` using the `this` keyword. To stop a callback from handling the *onSetFocus( )* event, use the *delete* operator, as in:

```
// Make sure to omit the () operator after the function name!
delete next_btn.onSetFocus;
```

To detect attainment of focus via either the keyboard or mouse, use the *onRollOver( )* handler, which executes both when the mouse is over `theButton` and when `theButton` gains keyboard focus. To capture all focus events centrally, rather than for a single instance, use *Selection.onSetFocus( )*.

To activate a keyboard-focused button, the user must press the Enter key or spacebar (which has the same effect as clicking the button and causes the button's *onRelease( )* handler to execute).

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

See Also

Button.onKillFocus( ), *Button.onRollOver( )*, *MovieClip.onSetFocus( )*, *Selection.onSetFocus( )*, *Selection.setFocus( )*, *TextField.onSetFocus( )*; Chapter 10

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Chapter 18. ActionScript Language Reference

Button._parent Property — Flash 6

a reference to the movie clip in which the button resides — *read/write*

theButton._parent

Description

The _parent property is an object reference to the movie clip that contains *theButton*. For example, if we have a button named go_btn in a clip named mainSection_mc, then go_btn. _parent refers to mainSection_mc.

Usage

Though it is possible to reassign the _parent property to another clip (or any other value), doing so has little purpose—only the reference is changed, not the physical structure of the clip and buttons.

Example

The _parent property is often used to control a movie clip from a button event handler, based on some user interaction. For example, in the following code, when begin_btn is clicked, the movie clip in which it resides plays:

```
begin_btn.onRelease = function () {
  this._parent.play();
}
```

Note that in the Flash 5-style of attaching event handlers directly to objects (e.g., *on (press)*), the _parent reference is not necessary, because this refers to the containing movie clip. For information on button event handler scope, see Chapter 10.

See Also

Button.onRelease( ), *MovieClip*._parent; Chapter 10

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Chapter 18. ActionScript Language Reference

Button._rotation Property — Flash 6

rotation, in degrees, of a button — ***read/write***

theButton._rotation

Description

The floating-point _rotation property specifies the number of degrees *theButton* is rotated about its registration point. Both author-time and runtime adjustments are reflected in _rotation. See *MovieClip*._rotation for details on various positive and negative values for rotation, including values that are out of range.

Example

The following code rotates a button when the button is clicked:

```
fun_btn.onRelease = function () {
  this._rotation += 10;
}
```

See Also

Button.onRelease( ), *MovieClip*._rotation

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Button.tabEnabled Property

Flash 6

Boolean; includes or excludes the button from the current tab order ***read/write***

theButton.tabEnabled

Description

The Boolean `tabEnabled` property governs whether *`theButton`* is part of the tab order for the text fields, movie clips, and buttons currently on screen. The *tab order* is the sequence in which objects are focused when the Tab key is pressed in a movie. See *`TextField`*`.tabEnabled` for complete details.

For buttons, the `tabEnabled` property can exclude a button with a defined `tabIndex` from the custom tab order, or it can exclude any button from the automatic tab order. A button is typically removed from the tab order when it is disabled (its `enabled` property is `false`) or when it should be accessible only to mouse input, such as in games that use transparent buttons to act as mouse hit regions.

A movie clip's `tabChildren` property controls whether the text fields, clips, and buttons it contains are included in the automatic tab order. To disable tabbing for a group of buttons quickly, put them in a movie clip and set its `tabChildren` to `false`.

Usage

The Tab key does not work reliably in Test Movie mode unless Control Disable Keyboard Shortcuts is enabled. Be sure to test keyboard behavior in the Standalone Player, if applicable, and in the Flash Player for all targeted web browsers.

Example

The following code prevents the keyboard from accessing `done_btn`:

```
done_btn.tabEnabled = false;
```

See Also

`Button``.enabled`, *`Button`*`.tabIndex`, *`MovieClip`*`.tabChildren`, *`MovieClip`*`.tabEnabled`, *`TextField`*`.tabEnabled`

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Chapter 18. ActionScript Language Reference

Button.tabIndex Property — Flash 6

specifies the button's index in the custom tab order — *read/write*

theButton.tabIndex

Description

The integer tabIndex property specifies *theButton*'s position in the custom tab order. When no text field, clip, or button has a tabIndex property defined, Flash determines the tab order automatically, and pressing the Tab key moves focus between buttons and text fields from right to left, then top to bottom. Use the tabIndex property to force Tab key navigation to follow a custom sequence, overriding the automatic order. For complete details, see *TextField*.tabIndex.

Objects with _visible set to false are not included in the tab order. When at least one object has tabIndex set, any object without an assigned tabIndex cannot be accessed via the Tab key.

By default, buttons are tab-enabled, so the tabEnabled property need only be set to exclude a button from the tab order while retaining its tabIndex value (allowing segments of a keyboard-controlled interface to be disabled and restored with minimal effort).

Usage

The Tab key does not work reliably in Test Movie mode unless Control Disable Keyboard Shortcuts is enabled. Be sure to test keyboard behavior in the Standalone Player, if applicable, and in the Flash Player for all targeted web browsers.

Example

The following code restricts Tab key access to three buttons, in this order: jump_btn, moveLeft_btn, moveRight_btn. The order applies regardless of the position of the buttons on screen.

```
jump_btn.tabIndex = 1;
moveLeft_btn.tabIndex = 2;
moveRight_btn.tabIndex = 3;
```

See Also

Button.tabEnabled, *TextField*.tabIndex, *MovieClip*.tabIndex

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Chapter 18. ActionScript Language Reference

Button._target Property Flash 6

the target path of the button, in slash syntax ***read/write***

theButton._target

Description

The _target property represents the path to *theButton* in a Flash 4-style slash notation string. For example, if go_btn resides on the main movie timeline, go_btn's _target property is "/go_btn". To retrieve a string that gives the path to a button in dot notation, use the global *targetPath( )* function. Because slash notation is deprecated, this property is not commonly used and is supported only for consistency with the legacy *MovieClip*._target property.

See Also

targetPath( ), *MovieClip*._target

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Chapter 18. ActionScript Language Reference

Button.trackAsMenu Property Flash 6

modifies the button's onRelease( ) handler requirements, enabling menu-style behaviorFlash *read/write* 3; property form introduced in

`theButton.trackAsMenu`

Description

The Boolean `trackAsMenu` property modifies the conditions that cause `theButton`'s *onRelease( )* event handler to execute. It is normally used to turn a group of buttons into a menu. When `trackAsMenu` is `undefined` (the default) or `false`, a button's *onRelease( )* handler will execute only when the mouse is both pressed *and* released over the button. When `trackAsMenu` is `true`, releasing the mouse over the button will always trigger its *onRelease( )* handler, unless the mouse was originally pressed over a button for which `trackAsMenu` is `false` or `undefined`. The following scenarios demonstrate the result of this subtle change in behavior.

Consider a "Products" menu button with three suboption buttons: Music, Books, and Software. These steps show how the menu works when `trackAsMenu` is `false` for all four buttons:

1. Products button is pressed.
2. Suboption buttons appear (are attached at runtime).
3. Mouse moves over first option, Music. The Music button's *Up* state is shown.
4. Mouse is released over Music. No *onRelease( )* handler is executed, because the mouse was pressed over Products but released over a different button, Music. User's selection is not activated.

These steps show how the menu works when `trackAsMenu` is `true` for all four buttons:

1. Products button is pressed.
2. Suboption buttons appear (are attached at runtime).
3. Mouse moves over first option, Music. The Music button's *Down* state is shown, revealing the user's current selection.
4. Mouse is released over Music. The Music button's *onRelease( )* handler is executed. The user's selection is activated.

Notice that as a side effect of `trackAsMenu` being `true`, the *Down* state of a button is always shown when the mouse pointer is over the button and the mouse button is down (even if the mouse was originally pressed on another button). The button's *onRollOver( )* and *onRollOut( )* handlers, however, are not triggered unless the mouse button is up.

The `trackAsMenu` property should be set the same for all buttons in a menu or other group-based interface; otherwise, mayhem and confusion can ensue.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Example

On frame 1 of a movie, create three button instances, named `b1`, `b2`, and `b3`, each with unique *Up*, *Over*, and *Down* states. Then attach the following code to frame 1 and choose Control Test Movie. Click the mouse over `b1` and release it over `b2`. Notice that `b2`'s Down state is displayed, and its *onRelease( )* handler executes.

```
b1.trackAsMenu = true;
b2.trackAsMenu = true;
b3.trackAsMenu = true;

b1.onRelease = b2.onRelease = b3.onRelease = release;

function release () {
  trace(this._name + " released");
}
```

See Also

Button.onRelease, *Button*.onRollOver, *Button*.onRollOut

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Chapter 18. ActionScript Language Reference

Button._url Property

Flash 6

---

the network address of the movie clip that contains the button

read-only

theButton._url

Description

The _url property represents the URL (Uniform Resource Locator) indicating the Internet or local disk location of *theButton*._parent, as a string. The _url property is always an absolute URL, such as "*http://www.moock.org/asdg/examples/someMovie.swf*", never a relative one. See *MovieClip*._url for details. Note that *theButton*._url is always the same as *theButton*._parent._url.

See Also

Button._parent, *MovieClip*._url

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Chapter 18. ActionScript Language Reference

Button.useHandCursor Property — Flash 6

Boolean; dictates whether a hand icon is displayed when the mouse pointer is over the button — ***read/write***

```
theButton.useHandCursor
```

Description

When set to `false`, the Boolean `useHandCursor` property disables the display of the hand pointer icon normally shown when the mouse pointer is over *`theButton`*'s hit area. When the hand cursor is disabled, the regular arrow cursor is used (this is more typical behavior for operating system buttons). Movies that use large transparent buttons, for interfaces such as picture slide shows, often disable the hand cursor.

Note that other than the hand icon, no other built-in mouse pointer icons are available. For instructions on creating a custom mouse pointer icon, see *Mouse.hide*( ).

Example

The following code disables the hand cursor for `ok_btn`:

```
ok_btn.useHandCursor = false;
```

The following code disables the hand cursor for all buttons in a movie (unless a button overrides the global setting with its own `useHandCursor` property):

```
Button.prototype.useHandCursor = false;
```

See Also

Mouse.hide( ); "The Almighty Prototype Chain," in Chapter 12

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Chapter 18. ActionScript Language Reference

Button._visible Property — Flash 6

whether the button is shown or hidden — ***read/write***

theButton._visible

Description

The Boolean _visible property indicates whether *theButton* is shown (true) or hidden (false). We use _visible as a quick means of hiding a button from view. Hidden buttons can still be controlled via ActionScript; they simply are not displayed on screen. However, a hidden button cannot receive focus or user input, so its events do not get triggered.

Button._visible is directly analogous to *MovieClip*._visible; see that property entry for full details.

Example

The following code hides quit_btn:

```
quit_btn._visible = false;
```

See Also

Button._alpha, *MovieClip*._visible

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Chapter 18. ActionScript Language Reference

Button._width Property — Flash 6

width of the button, in pixels — ***read/write***

theButton._width

Description

The floating-point _width property stores a nonnegative number specifying the current width of *theButton*, in pixels. If *theButton* has no content, _width is 0. The _width property measures the content of a button as the distance between the leftmost occupied pixel and the rightmost occupied pixel, even if there is empty space between those pixels. By setting _width, we change the size of *theButton*, affecting all of its states (its *Up*, *Over*, and *Down* frames). Note that the _width of a button can change as it switches between these states.

Button._width is directly analogous to *MovieClip*._width; see that property entry for full details.

Example

```
// Check start_btn's width
trace(start_btn._width);

// Set start_btn's width to 200
start_btn._width = 200;

// Create a text field 10 pixels to the right of start_btn, assuming
// that start_btn's top left corner is at the registration point
start_btn._parent.createTextField("theField_txt",
                                  1,
                                  start_btn._x + start_btn._width + 10,
                                  start_btn._y,
                                  200, 20);
theField_txt.border = true;
theField_txt.text = "Click the button to start.";
```

See Also

Button._height, *MovieClip*._width

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Chapter 18. ActionScript Language Reference

Button._x Property Flash 6

horizontal location of the button, in pixels *read/write*

theButton._x

Description

The floating-point _x property indicates the horizontal position of *theButton* 's registration point. It is measured relative to one of two possible coordinate spaces:

- If *theButton* resides on the main timeline, _x is measured relative to the *Stage's left edge* .
- If *theButton* resides on a movie clip instance's timeline, _x is measured relative to the *registration point* of that parent instance.

The _x property (along with all horizontal coordinates in Flash) increases to the right and decreases to the left.

Button ._x is directly analogous to *MovieClip* ._x ; see that property entry for full details.

Example

```
// Place quit_btn 100 pixels to the right of its parent's registration point
quit_btn._x = 100;
// Move the button over 10 more pixels to the right
quit_btn._x += 10;
```

See Also

Button ._y,*MovieClip* ._x

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Chapter 18. ActionScript Language Reference

Button._xmouse Property — Flash 6

horizontal location of mouse pointer, relative to the button's registration point — ***read-only***

theButton._xmouse

Description

The floating-point _xmouse property indicates the horizontal location of the mouse pointer's hotspot, relative to *theButton*'s registration point. To obtain a consistent _xmouse coordinate that is always measured relative to the Stage, use _root._xmouse.

See Also

Button._ymouse, *MovieClip*._xmouse

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Chapter 18. ActionScript Language Reference

Button._xscale Property Flash 6

width of a button, as a percentage ***read/write***

theButton._xscale

Description

The floating-point _xscale property specifies the width of *theButton*, relative to its original width, expressed as a percentage. Its "original width" is the pixel width of its Library symbol. When the current width of *theButton* is the same as its original width, _xscale is 100. An _xscale of 200 doubles *theButton*'s original width; an _xscale of 50 halves its original width.

Button._xscale is directly analogous to *MovieClip*._xscale; see that property entry for full details.

Example

```
// Double run_btn's width, relative to its Library symbol's width
run_btn._xscale = 200;
```

See Also

Button._width, *Button*._yscale, *MovieClip*._xscale

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Chapter 18. ActionScript Language Reference

Button._y Property — Flash 6

vertical location of the button, in pixels — ***read/write***

theButton._y

Description

The floating-point _y property indicates the vertical position of *theButton*'s registration point. It is measured relative to one of two possible coordinate spaces:

- If *theButton* resides on the main timeline, _y is measured relative to the *Stage's top edge*.
- If *theButton* resides on a movie clip instance's timeline, _y is measured relative to the *registration point* of that parent instance.

The _y property (along with all vertical coordinates in Flash) increases downward and decreases upward—the opposite of the Cartesian coordinate system.

Button._y is directly analogous to *MovieClip*._y; see that property entry for full details.

Example

```
// Place quit_btn directly on its parent's registration point.
quit_btn._x = 0;
quit_btn._y = 0;
```

See Also

Button._x, *MovieClip*._y

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Chapter 18. ActionScript Language Reference

Button._ymouse Property — Flash 6

vertical location of mouse pointer, relative to the button's registration point — ***read-only***

theButton._ymouse

Description

The floating-point _ymouse property indicates the vertical location of the mouse pointer's hotspot, relative to *theButton*'s registration point. To obtain a consistent _ymouse coordinate that is always measured relative to the Stage, use _root._ymouse.

See Also

Button._xmouse, *MovieClip*._ymouse

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Chapter 18. ActionScript Language Reference

Button._yscale Property — Flash 6

height of a button, as a percentage — ***read/write***

theButton._yscale

Description

The floating-point _yscale property specifies the height of *theButton*, relative to its original height, expressed as a percentage. Its "original height" is the pixel height of its Library symbol. When the current height of *theButton* is the same as its original height, _yscale is 100. A _yscale of 300 triples *theButton*'s original height; a _yscale of 50 halves its original height.

Button._yscale is directly analogous to *MovieClip*._yscale; see that property entry for full details.

Example

```
// Restore sort_btn to its original (Library symbol) size
sort_btn._yscale = 100;
sort_btn._xscale = 100;
```

See Also

Button._height, *Button*._xscale, *MovieClip*._yscale

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

call( ) Global Function — Flash 4

execute the script of a remote frame

```
call(frameLabel)
call(frameNumber)
```

Arguments

frameLabel

A string representing the label of the frame whose script should be executed.

frameNumber

The number of the frame whose script should be executed.

Description

The *call( )* function executes the script attached to the frame at `frameLabel` or `frameNumber`. For example, the following code runs the script on frame 20 of the current timeline:

```
call(20);
```

In Flash 4, *call( )* was used to create a crude kind of reusable subroutine (one that could not accept parameters or return a value). As of Flash 5, the *function* statement is the preferred way to define a function.

Note that in Flash 5 and later, when a script is executed remotely via *call( )*, any variables declared with the *var* keyword are considered local to that execution and expire after the script completes. To create nonlocal variables in a remotely-executed script, omit the *var* keyword:

```
var x = 10;  // Local variable; dies after script completes
x = 10;      // Timeline variable; persists after script completes
```

To invoke *call( )* on frames outside the current timeline, use the deprecated *tellTarget( )* function. The following example executes the script on frame 10 of the `box` clip:

```
tellTarget ("box") {
  call(10);
}
```

See Also

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Chapter 9; Appendix C

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Chapter 18. ActionScript Language Reference

Capabilities Object — Flash 6

information about the Flash Player and its host system

Properties

hasAccessibility

Indicates whether the Player supports accessibility (MSAA).

hasAudio

Indicates whether the Player supports audio playback.

hasAudioEncoder

Indicates whether the system can encode audio.

hasMP3

Indicates whether the Player supports MP3 sound playback.

hasVideoEncoder

Indicates whether the system can encode video.

input

The user input device for the system, such as a mouse (indicated by the word "point").

isDebugger

Indicates whether the Player is a debugging version.

language

The current language setting on the operating system.

manufacturer

The creator of the Player.

os

The operating system on which the Player is running.

pixelAspectRatio

The width-to-height ratio of a pixel on the screen.

screenColor

The color mode supported by the screen (color, grayscale, or black and white).

screenDPI

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

The number of pixels per inch displayed by the screen.

screenResolutionX

The width of the screen, in pixels.

screenResolutionY

The height of the screen, in pixels.

serverString

All *Capabilities* property values, as a string of URL-encoded variables.

version

The Flash Player version.

Description

The *Capabilities* object represents the specifications of the Flash Player and the system on which the Player is running. It is stored as a property of the *System* object (i.e., `System.capabilities`). Hence, references to *Capabilities* properties always start with the *System* object, as follows:

```
System.capabilities.propertyName
```

For example, the following code displays the screen width in the Output window:

```
trace(System.capabilities.screenResolutionX);
```

The *Capabilities* object is used to write code that adapts a movie to its playback environment. It applies primarily to mobile devices, such as Pocket PCs and cell phones. For example, by querying the value of `hasAudio`, a movie playing on a handheld device can check if the device can play audio. If `hasAudio` is `false`, the movie can provide text subtitles.

All properties of *Capabilities* have corresponding short-form names and values that can be used to send the client-side Player abilities to a server-side script. For example, the `screenColor` property's short form is `COL`, with possible values of "color", "gray", and "bw". The `capabilities.serverString` property joins all short-form property name/value pairs into a URL-encoded string suitable for transmission to a server. A server can read this information and respond by sending the appropriate movie or data for the playback environment. See `capabilities.serverString` for details.

For information on authoring Flash movies for devices, see:

http://www.macromedia.com/software/flashplayer/resources/devices/
http://www.flashenabled.com/mobile/

For a list of devices that the Flash Player runs on, see:

http://www.macromedia.com/software/flashplayer/resources/devices/supported_devices/

Usage

Though the properties of *Capabilities* are writable, there is no practical reason to overwrite their system-assigned values. Doing so obscures the information they contain. Custom assignments to properties other than `serverString` are not reflected in the value of `serverString`.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

See Also

The *System* object, the *Stage* object

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Chapter 18. ActionScript Language Reference

capabilities.hasAccessibility Property — Flash 6

indicates whether the Player supports Microsoft Active Accessibility (MSAA) — *read/write*

`System.capabilities.hasAccessibility`

Description

In Flash Player 6, the Boolean `hasAccessibility` property is `true` if the Player supports accessibility. At the time of this writing, only the Windows ActiveX version of Flash Player 6 supports accessibility (via Microsoft Active Accessibility, MSAA). MSAA is a Microsoft API that exposes an application's user interface elements to screen readers and other accessibility aids, such as GW Micro's Window-Eyes (http://www.gwmicro.com).

Movies that rely on accessibility aids should check that both `hasAccessibility` and *Accessibility.isActive( )* return `true` before sending any accessibility information. The `hasAccessibility` property tells us that the Player supports accessibility, but the *Accessibility.isActive( )* method tells us that a screen reader is actually running and communicating with the Flash Player.

For more information on MSAA, see:

http://msdn.microsoft.com/library/en-us/msaa/msaastart_9w2t.asp

See the *Accessibility* object for more details on accessibility.

The corresponding server string for `hasAccessibility` is `ACC`, with possible values of `t` (true) and `f` (false). See `capabilities.serverString` for details.

See Also

The *Accessibility* object

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Chapter 18. ActionScript Language Reference

capabilities.hasAudio Property — Flash 6

indicates whether the Player supports audio playback — ***read/write***

```
System.capabilities.hasAudio
```

Description

The Boolean `hasAudio` property is `true` for Flash Players that support audio playback, and `false` for Players that cannot play sounds. While all desktop versions of Flash support sound, not all handheld or set-top device versions necessarily do. Movies can use this property to provide alternative content, such as subtitles, when sound is not available.

Note that `hasAudio` indicates only whether the Flash Player supports audio playback, not whether the computer actually has audio enabled (i.e., has a sound card, speakers turned on, volume on, etc.).

The corresponding server string for `hasAudio` is `A`, with possible values of `t` (true) and `f` (false). See `capabilities.serverString` for details.

See Also

`capabilities.hasAudioEncoder`, `capabilities.hasMP3`

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Chapter 18. ActionScript Language Reference

capabilities.hasAudioEncoder Property Flash 6

indicates whether the system can encode audio ***read/write***

```
System.capabilities.hasAudioEncoder
```

Description

The Boolean `hasAudioEncoder` property indicates whether the Player can encode an audio stream (for example, coming from a microphone), normally for transmission to a server such as Macromedia Flash Communication Server MX (Comm Server). In the desktop version of Flash Player 6, `hasAudioEncoder` is always `true`, indicating that audio encoding is available. This does not guarantee, however, that the necessary hardware is present.

The corresponding server string for `hasAudioEncoder` is `AE`, with possible values of `t` (true) and `f` (false). See `capabilities.serverString` for details.

See Also

`capabilities.hasAudio`, `capabilities.hasVideoEncoder`

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Chapter 18. ActionScript Language Reference

capabilities.hasMP3 Property — Flash 6

indicates whether the Player supports MP3 sound playback — ***read/write***

`System.capabilities.hasMP3`

Description

The Boolean `hasMP3` property is `true` for Flash Players that can play MP3 audio files, and `false` for those without MP3 support. It typically is used to determine the appropriate type of sound file to load at runtime. For example, a movie that detects a system with audio support (via `hasAudio`), but no MP3 support, could load a regular ADPCM sound.

The corresponding server string for `hasMP3` is `MP3`, with possible values of `t` (true) and `f` (false). See `capabilities.serverString` for details.

See Also

`capabilities.hasAudio`

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Chapter 18. ActionScript Language Reference

capabilities.hasVideoEncoder Property Flash 6

indicates whether the system can encode video ***read/write***

```
System.capabilities.hasVideoEncoder
```

Description

The Boolean `hasVideoEncoder` property indicates whether the Player can encode a video stream (for example, coming from a webcam), normally for transmission to a server such as Macromedia Flash Communication Server MX (Comm Server). In the desktop version of Flash Player 6, `hasVideoEncoder` is always `true`, indicating that video encoding is available. This does not guarantee, however, that the necessary hardware (such as a webcam) is present.

The corresponding server string for `hasVideoEncoder` is `VE`, with possible values of `t` (true) and `f` (false). See `capabilities.serverString` for details.

See Also

`capabilities.hasAudioEncoder`

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Chapter 18. ActionScript Language Reference

capabilities.input Property Flash 6

the user input device for the system ***read/write***

```
System.capabilities.input
```

Description

The string `input` property indicates the type of input device provided by the system hardware. For mouse-style devices, the type is "point". Other names for input devices, such as a stylus or a cursor pad on a cell phone, will be determined as they become available. The `input` property allows a movie to switch to, say, a custom made keypad-only input mode, which would be much more limited than the typical mouse input mode.

The corresponding server string for `input` is `I`, with possible values of `point` and other input device names to be determined. See `capabilities.serverString` for details.

See Also

The *Mouse* object

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Chapter 18. ActionScript Language Reference

capabilities.isDebugger Property — Flash 6

indicates whether the Player is a debugging version — ***read/write***

`System.capabilities.isDebugger`

Description

The Boolean `isDebugger` property indicates whether the Player is the release version (`false`) or the debugging version (`true`). The debugging version of the Player is used in conjunction with the Flash authoring tool to debug Flash movies. This property is not normally needed by Flash developers.

The corresponding server string for `isDebugger` is `DEB`, with possible values of `t` (true) and `f` (false). See `capabilities.serverString` for details.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Chapter 18. ActionScript Language Reference

capabilities.language Property — Flash 6

the current language setting on the operating system — ***read/write***

`System.capabilities.language`

Description

The string `language` property contains a two-letter code representing the language used for text on the operating system. The cross-platform codes used are a subset of those in the ISO-639-1 standard, "Codes for the Representation of Names of Languages" (see http://lcweb.loc.gov/standards/iso639-2/englangn.html). For example, "en" indicates English, and "fr" indicates French. Where applicable, `language` also includes a two-letter country code indicating the local variant of the language, as defined by ISO 3166 (see http://www.iso.org/iso/en/prods-services/iso3166ma/02iso-3166-code-lists/list-en1.html). A hyphen separates the language code from the country code, if both are present. For example, "en-UK" indicates English, United Kingdom variant.

By checking the `language` setting, a movie can intelligently provide text in the appropriate language for the user's system. Multiple language assets can be embedded directly into a single movie or loaded dynamically as variables, XML, or entire movies.

The corresponding server string for `language` is `LAN`, with possible values listed in Table 18-4. See `capabilities.serverString` for details.

Table 18-4. Flash language codes

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Language	ISO 639-1 language code	ISO 3166 country code
English, United States	en	US
English, United Kingdom	en	UK
French	fr	
Korean	ko	
Japanese	ja	
Swedish	sv	
German	de	
Spanish	es	
Italian	it	
Simplified Chinese	zh	CN
Traditional Chinese	zh	TW
Portuguese	pt	
Polish	pl	
Hungarian	hu	
Czech	cs	
Turkish	tr	
Finnish	fi	
Danish	da	
Norwegian	no	
Dutch	nl	
Russian	ru	
Other/Unknown	xu	

Bugs

The English language country codes (UK and US) are not properly set by all versions of Flash Player 6; In both countries, `System.capabilities.language` returns "en" and does not include a country code.

Example

The following code loads the English or French version of a movie, depending on the current language setting:

```
if (System.capabilities.language.substring(0,2) =  = "en") {
  trace("English language");
  content_mc.loadMovie("news-english.swf");
} else if (System.capabilities.language =  = "fr") {
  trace("French language");
  content_mc.loadMovie("news-french.swf");
}
```

See Also

Character Encoding," in Chapter 4

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Chapter 18. ActionScript Language Reference

capabilities.manufacturer Property Flash 6

indicates the manufacturer of the Flash Player ***read/write***

```
System.capabilities.manufacturer
```

Description

The string `manufacturer` property indicates the name of the organization that created the Flash Player. Third parties can license Macromedia's Flash Player source code and port the Player to new platforms or devices. Developers of ported or custom Players can provide a custom value for the `manufacturer` property. For the desktop version of the Flash Player, the only values are "Macromedia Windows" and "Macromedia Macintosh". Do not rely on `manufacturer` to provide operating system or version information. Use `capabilities.os` and `capabilities.version` instead.

The `manufacturer` property can be used to react to special support for custom capabilities implemented by a specific Player manufacturer.

For further information, see:

http://www.macromedia.com/software/flash/open/licensing/

The corresponding server string for `manufacturer` is `M`, with values determined by individual manufacturers. See `capabilities.serverString` for details.

See Also

`capabilities.os`, `capabilities.version`

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Chapter 18. ActionScript Language Reference

capabilities.os Property — Flash 6

indicates the operating system on which the Player is running — ***read/write***

```
System.capabilities.os
```

Description

The string `os` property indicates the name of the operating system on which the Player is running. It provides more granular information than the *getVersion( )* function, which does not distinguish between OS varieties such as "Windows 2000" and "Windows 98". The current values for `os` on Windows are:

Windows XP
Windows 2000
Windows NT
Windows 98/ME
Windows 95
Windows CE

On Macintosh, `os` returns "MacOS *x*.*y*.*z*", where *x* is the major release, *y* is the minor release, and *z* is the dot release. For example:

MacOS 9.2.1
MacOS 10.1.4

The corresponding server string for `os` is `OS`, with the preceding values listed for Windows or Macintosh or determined by individual manufacturers. See `capabilities.serverString` for details.

Bugs

Macromedia's documentation for Flash MX incorrectly claims that `os` defaults to an empty string. It defaults to the value appropriate for the OS under which the Player is running.

See Also

`capabilities.version`, `capabilities.manufacturer`

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Chapter 18. ActionScript Language Reference

capabilities.pixelAspectRatio Property Flash 6

the width-to-height ratio of a pixel on the screen ***read/write***

```
System.capabilities.pixelAspectRatio
```

Description

The floating-point `pixelAspectRatio` property indicates the proportions of a single pixel on the screen. It is the ratio of a pixel's width to its height. For example, on screens where pixels are half as wide as they are high, `pixelAspectRatio` will return .5 (1/2). On most desktop computers, pixels are square, so `pixelAspectRatio` is 1.

The `pixelAspectRatio` property lets a movie compensate for potential distortion caused by the aspect ratio of the pixels on the playback device. For example, a circle with a constant radius will appear as an ellipse on devices whose pixel aspect ratio is not 1. On a device with pixels twice as wide as they are high, you might load an alternate movie authored at half of the original width so that the final output displays correctly. At the time of this writing, `pixelAspectRatio` always returns 1, and the development process for nonsquare pixel devices is not yet determined. Always test on the target device to ensure correct appearance at a nonunity aspect ratio. More information will be provided by Macromedia when it becomes relevant.

The corresponding server string for `pixelAspectRatio` is `AR`, with a value of a floating-point number, such as 1.0.

Bugs

Macromedia's documentation for Flash MX incorrectly claims that the server string for `pixelAspectRatio` is `PAR`. The correct server string is `AR`.

See Also

`capabilities.screenDPI`, `capabilities.screenResolutionX`, `capabilities.screenResolutionY`

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Chapter 18. ActionScript Language Reference

capabilities.screenColor Property — Flash 6

the color mode supported by the screen (color, grayscale, or black and white) — ***read/write***

`System.capabilities.screenColor`

Description

The string `screenColor` property indicates the color mode supported by the device on which the Flash Player is running. It allows movies with subtle color differences to switch to a high-contrast color scheme when played on a grayscale or black and white system. Possible values are: "color" (supports more than two colors), "gray" (grayscale), and "bw" (black and white or 2-color display).

The corresponding server string for `screenColor` is `COL`, with possible values of "color", "gray", and "bw". See `capabilities.serverString` for details.

See Also

The *Color* class

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Chapter 18. ActionScript Language Reference

capabilities.screenDPI Property — Flash 6

the number of pixels per inch displayed by the screen, typically 72 dpi — ***read/write***

```
System.capabilities.screenDPI
```

Description

The integer `screenDPI` property indicates the number of dots (pixels) per inch displayed by the screen. Typical desktop computer monitors display at 72 pixels per inch. Small handheld devices may display much lower or higher pixels per inch. Relatively speaking, the higher the `screenDPI`, the smaller elements in a movie will appear.

Though two screens may have the same horizontal and vertical resolution, the actual display size of a movie can vary greatly according to the physical dimensions of the screen and the number of pixels per inch. For example, consider two screens, each with a resolution of 100 x 100. If one screen is 2 inches square (50 pixels per inch) and the other is 5 inches square (20 pixels per inch), then a 10-pixel font will be one-fifth of an inch high on the first screen, and half an inch high on the second. Depending on the differences between screens, text can easily be readable on one screen and illegible on another. By checking the `screenDPI`, `screenResolutionX`, and `screenResolutionY` properties, a movie can scale its contents to be legible on the given display.

The corresponding server string for `screenDPI` is `DP`, with a possible value of any integer. See `capabilities.serverString` for details.

Usage

Print designers will recognize the use of the acronym DPI (*dots per inch*) as a slight misnomer. Canonically, "DPI" applies to the printed page, while "PPI" (*pixels per inch*) applies to the screen. The imprecision, however, is here to stay. Note that it is pixels per linear inch, not pixels per square inch.

See Also

`capabilities.pixelAspectRatio`, `capabilities.screenResolutionX`, `capabilities.screenResolutionY`, `Stage.scaleMode`

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Chapter 18. ActionScript Language Reference

capabilities.screenResolutionX Property — Flash 6

the width of the screen, in pixels — ***read/write***

```
System.capabilities.screenResolutionX
```

Description

The integer `screenResolutionX` property returns the total width, in pixels, of the screen on which the Flash Player is displayed. It is analogous to JavaScript's `Screen.width` property. You can use the `screenResolutionX` to center a browser pop-up window or inform the user when their resolution is too small to display a movie, as shown in the Example.

If we know the physical dimensions of a screen, we can use `screenResolutionX` and `screenResolutionY` in combination with `screenDPI` to calculate the actual size of an element in a movie. This is useful for scaling text to appear at the same size on all displays, as discussed under `capabilities.screenDPI`.

To retrieve the width of a movie's Stage rather than the whole screen or monitor, use `Stage.width`.

The corresponding server string for `screenResolutionX` is `R`, with a possible value that includes both the width and height of the screen, formatted as "*WIDTH*x*HEIGHT*" (for example, "1600x1200").

Example

The following code displays a warning when the screen resolution is too small:

```
// Specify required resolution
minResolutionX = 800;
// If the requirement is not met...
if (System.capabilities.screenResolutionX < minResolutionX) {
  // ...display a warning message
  this.createTextField("warning_txt", 1, 300, 300, 250, 30);
  this.warning_txt.border = true;
  this.warning_txt.text = "Movie requires an 800-pixel wide screen.";

  // Display a disabled frame
  this.gotoAndStop("movieDisabled");
}
```

Bugs

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

In the first release of Macromedia's documentation for Flash MX, this property is incorrectly listed as `System.capabilities.screenResolution.x`. There should be no period before the final `x`, and the `X` should be capitalized.

See Also

`capabilities.pixelAspectRatio`, `capabilities.screenDPI`, `capabilities.screenResolutionY`, `Stage.width`

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Chapter 18. ActionScript Language Reference

capabilities.screenResolutionY Property Flash 6

the height of the screen, in pixels ***read/write***

`System.capabilities.screenResolutionY`

Description

The integer `screenResolutionY` property returns the total height, in pixels, of the screen on which the Flash Player is displayed. It is analogous to JavaScript's `Screen.height` property. To retrieve the height of a movie's Stage rather than the whole screen, use `Stage.height`.

The corresponding server string for `screenResolutionY` is `R`, with a possible value that includes both the width and height of the screen, formatted as "*WIDTH*x*HEIGHT*" (for example, "800x600"). See `capabilities.screenResolutionX` for an example that checks the screen resolution.

Bugs

In the first release of Macromedia's documentation for Flash MX, this property is incorrectly listed as `System.capabilities.screenResolution.y`. There should be no period before the final `y`, and the `Y` should be capitalized.

See Also

`capabilities.pixelAspectRatio`, `capabilities.screenDPI`, `capabilities.screenResolutionX`, `Stage.height`

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Chapter 18. ActionScript Language Reference

capabilities.serverString Property — Flash 6

all Capabilities object property values, as a string of URL-encoded variables — ***read/write***

`System.capabilities.serverString`

Description

The string `serverString` property contains a list of URL-encoded variable names and values corresponding to the property names and values of the *Capabilities* object.

For example, on my system, the value of `serverString` is:

```
A=t&MP3=t&AE=t&VE=t&ACC=f&DEB=t&V=WIN%206%2C0%2C21%2C0&M=Macromedia Windows&
R=1600x1200&DP=72&COL=color&AR=1.0&I=point&OS=Windows XP&L=en
```

To determine the variable name and value used for each property, see the entry for that property. The `serverString` property is provided as a convenience to the developer who wishes to retrieve assets from the server, based on the client-side Player specifications. The string can be used when loading external XML, variables, or even other movies. For example, a simple movie, *loadStub.swf* , could be used as a loading module for subsequent content. The *loadStub.swf* movie should issue a *loadMovie( )* command to a server script and append the `serverString` property value to a GET request:

```
content_mc.loadMovie("content.swf?" + System.capabilities.serverString);
```

The server would then respond by sending the appropriate movie for the client-side environment.

Usage

Though the properties of the *Capabilities* object are writable, custom property assignments are not reflected in the value of `serverString` .

See Also

MovieClip.loadMovie( ) , the *LoadVars* class, the *XML* class

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Chapter 18. ActionScript Language Reference

capabilities.version Property — Flash 6

the Flash Player version — ***read/write***

`System.capabilities.version`

Description

The string `version` property contains version and platform information for the Flash Player hosting the current movie. of Flash 6, it is favored over the *getVersion( )* function, though the two return the same value. Because `capabilites.version` is not available in Flash 5, *getVersion( )* must be used for movies that will be played in Fla Player 5. Similarly, `$version` must be used for movies played in Flash Player 4.

The string stored in `version` takes the form:

```
platform majorVersion,minorVersion,buildNumber,patch
```

where `platform` is a code indicating the platform (`"WIN"` , `"MAC"` , or `"UNIX"`), followed by the major version number, the minor version number, and the build (a.k.a. revision) number. The last item, `patch` , is typically 0. For example:

```
WIN 5,0,30,0   // Version 5.0, Build 30 (5.0r30) on Windows
MAC 5,0,41,0   // Version 5.0, Build 41 (5.0r41) on Macintosh
UNIX 4,0,12,0  // Version 4.0, Build 12 (4.0r12) on Unix
```

To obtain a more detailed description of the current operating system, use `capabilities.os` .

In Test Movie mode, `version` reports the version number of the *Player* embedded in the authoring tool (which is not th same as the version of the authoring tool itself). For example, the Flash 5 authoring tool embeds the 5.0 r30 version of the Player, so it has a version of:

```
WIN 5,0,30,0
```

or:

```
MAC 5,0,30,0
```

Any time a major or minor version of the authoring tool is created, the `buildNumber` restarts at 0. However, in the typ development cycle of the Flash authoring tool, many builds of the Flash Player are produced before the final version of th authoring tool is released. Hence, the build number of the first new major version of a Player is usually greater than 0. For example, Flash Player 5 was first officially released at Build 30. Flash Player 6 was officially released at Build 23. A mov running in the first publicly released Flash Player 6 on Windows has a `version` of WIN 6,0,23,0.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Typically, we're concerned only with the platform, the major version, and the build number of a Player. To extract the por of the `version` string we're after, we can use the string manipulation tools described in Chapter 4, or we can construct a custom object with each component of the `version` string assigned to a property of that object, as shown in the followi Example.

When a movie is embedded in the browser, JavaScript and VBScript provide external tools for version detection, browser sniffing, and automatic page redirection. For details on detecting the Flash Player's presence and version with JavaScript a VBScript, see:

http://www.moock.org/webdesign/flash/detection/moockfpi

The corresponding server string for `version` is `V` , with possible values as described earlier in this Description.

Example

The following code extracts the various portions of the `version` string and stores them as the properties of an object fo access:

```
// Split up the version string into usable pieces.
var version = System.capabilities.version;
var firstSpace = version.indexOf(" ");
var tempString = version.substring(firstSpace + 1, version.length);
var tempArray = tempString.split(",");

// Assign the various parts of the getVersion( ) string to our object.
// Note that we convert the version number portions to integers.
var thePlayer = new Object();
thePlayer.platform = version.substring(0,firstSpace);
thePlayer.majorVersion = parseInt(tempArray[0]);
thePlayer.minorVersion = parseInt(tempArray[1]);
thePlayer.build = parseInt(tempArray[2]);
thePlayer.patch = parseInt(tempArray[3]);

// Now use our object to perform version-specific code.
if ((thePlayer.majorVersion =  = 6) && (thePlayer.build <= 23))  {
  this.createTextField("warning_txt", 1, 300, 300, 250, 30);
  this.warning_txt.border = true;
  this.warning_txt.text = "Please upgrade your Player to avoid streaming bug
}
```

Bugs

Macromedia's documentation for Flash MX incorrectly claims that the server string for `version` is `VER` (it actually is `V` that the value of `version` is an integer (it actually is a string).

See Also

`capabilities.os` , *getVersion( )* , `$version`

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Chapter 18. ActionScript Language Reference

clearInterval( ) Global Function Flash 6

stop the periodic execution of a function or method

```
clearInterval(intervalID)
```

Arguments

intervalID

An integer interval identifier returned by an earlier call to *setInterval( )*.

Description

The *setInterval( )* and *clearInterval( )* functions are used to create and destroy *intervals*, which execute a specified function or object method periodically. An interval is created by calling *setInterval( )*, which returns a unique identifier that can later be passed to *clearInterval( )* to halt the interval. An interval can be cleared either inside or outside of the executing interval function.

For examples and further details, see *setInterval( )*.

Usage

There is no JavaScript-style *setTimeout( )* function in ActionScript. To emulate *setTimeout( )*, create an interval function that clears itself the first time it runs, as shown under the Example for *setInterval( )*.

See Also

setInterval( )

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Chapter 18. ActionScript Language Reference

Color Class — Flash 5

control over movie clip color values

Constructor

`new Color(`*`target`*`)`

Arguments

target

A string or reference indicating the path to the movie clip or document level whose color will be controlled by the new object (references are converted to paths when used in a string context).

Methods

getRGB( )

Retrieve the current offset values for Red, Green, and Blue.

getTransform( )

Retrieve the current offset and percentage values for Red, Green, Blue, and Alpha.

setRGB( )

Assign new offset values for Red, Green, and Blue, while reducing percentage values to 0.

setTransform( )

Assign new offset and/or percentage values for Red, Green, Blue, and Alpha.

Description

We use objects of the *Color* class to dictate the color and transparency of a movie clip or main movie programmatically. Once we've created an object of the *Color* class for a specific `target`, we can then invoke the methods of that *Color* object to affect its `target`'s color and transparency. That is, there is no `color` property used to manipulate a movie clip's color; instead we create a *Color* object that targets the desired movie clip. For example, suppose we have a clip instance named `ball` that we want to color red. We first make a *Color* object with a `target` of `ball`, and store it in variable `ballColor`. Then we can use *ballColor.setRGB( )* to assign `ball` the color red, as follows:

```
var ballColor = new Color("ball");
ballColor.setRGB(0xFF0000);  // Pass setRGB( ) the hex value for red
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

For a custom example that adds a color-setting method directly to the *MovieClip* class, see the *Color.setRGB( )* method.

The preceding example provides color control for simple applications. But to handle more complex scenarios, we need to know more about how color is represented in Flash. Every individual color displayed in a movie clip is defined by four separate components: Red, Green, Blue, and Alpha (or transparency). These four components are combined in different amounts to generate each color we see on screen. The amount of Red, Green, Blue, and Alpha in a given color is describe as a number between 0 and 255. The higher the value of Red, Green, or Blue, the more each of those colors contributes to the final color. However, remember that computer color is additive, not subtractive like paint, so higher values tend to be brighter, not darker. If all three RGB components are equal, the result is a shade of gray; if they are all 0, the result is black; if they are all 255, the result is white. The higher the value of Alpha, the more opaque the final color will be. (A color with an Alpha of 0 is completely transparent, and a color with an Alpha of 255 is completely opaque.)

For example, pure red is described by the following values:

```
Red: 255, Green: 0, Blue: 0, Alpha: 255
```

whereas a partially transparent red might have the values:

```
Red: 255, Green: 0, Blue: 0, Alpha: 130
```

For the purposes of this discussion, we adopt the so-called *RGB triplet* notation (R: *red* , G: *green* , B: *blue*) when talking about color values. Although ActionScript doesn't support decimal RGB triplets such as (255, 0, 255), it does support the hexadecimal equivalent form `0x RRGGBB` , where `RR` , `GG` , and `BB` are each two-digit hex numbers representing Re Green and Blue. For example, the RGB triplet (R:51, G:51, B:102) is equivalent to the hexadecimal value `0x333366` .

We'll also adopt the so-called *RGBA quadlet* notation (R: *red* , G: *green* , B: *blue* , A: *alpha*) for convenience during the following discussion. Note that although there is no `color` property for a movie clip, clips do have an `_alpha` property that represents the alpha channel setting.

The initial Red, Green, Blue, and Alpha values for *each* color in a movie clip are set in the Flash authoring tool using the Color Mixer panel. (In the Color Mixer panel, Alpha is shown as a percentage, not a number from 0 to 255.) To alter *all* colors in a movie clip via ActionScript, we make universal adjustments (known as *transformations*) to the Red, Green, Blue, and Alpha components of the clip's color. Transformations can be applied at authoring time via the Property inspect (Flash MX) or Effect panel (Flash 5).

We have two means of setting transformations for each color component:

- We can set the *percentage* of the component's original value to a number between -100 and 100. For example, we c say, "set all red used in this clip to 80% of its original value." Although negative percentages aren't meaningful in isolation, they are used to calculate the final transformation as shown later in this Description.

- We can specify an amount to *offset* the component's original value. The offset is a number between -255 and 255. F example, we can say, "add 20 to all blue values in this clip," or, using a negative number, we can say, "subtract 30 from all blue values in this clip."

The final value of a color in a transformed clip is determined by combining its original (author-time) color component val with the transformation percentages and offsets set through the *Color* object, as follows:

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

```
R = originalRedValue   * (redTransformPercentage/100)   + redTransformOffset
G = originalGreenValue * (greenTransformPercentage/100) + greenTransformOffs
B = originalBlueValue  * (blueTransformPercentage/100)  + blueTransformOffse
A = originalAlphaValue * (alphaTransformPercentage/100) + alphaTransformOffs
```

If no transformations have been performed through ActionScript, the initial transformation *percentage* for each componen defaults to 100, while the initial *offset* defaults to 0.

Let's look at how color transformations work with a practical example. Suppose that a clip contains an opaque red triangle (R:255, G:0, B:0, A:255) and an opaque green circle (R:0, G:255, B:0, A:255). Also suppose that we apply a universal transformation to the clip, setting the percentage of Green to 50, the percentage of Alpha to 80, and the offset of Blue to 100, but leaving the other offsets and percentages at their defaults (0 or 100). Here's how the universal transformation affects our red triangle:

```
R =  = 255 * (100/100) +   0 =  = 255          // No change to Red
G =  =   0 *  (50/100) +   0 =  = 0            // Green reduced to 50%
B =  =   0 * (100/100) + 100 =  = 100          // Blue offset by 100
A =  = 255 *  (80/100) +   0 =  = 204          // Alpha reduced to 80%
```

The final transformed red triangle has the color value (R:255, G:0, B:100, A:204).

Now here's how the transformation affects our green circle:

```
R =  =   0 * (100/100) +   0 =  = 0            // No change to Red
G =  = 255 *  (50/100) +   0 =  = 127.5        // Green reduced to 50%
B =  =   0 * (100/100) + 100 =  = 100          // Blue offset by 100
A =  = 255 *  (80/100) +   0 =  = 204          // Alpha reduced to 80%
```

The final transformed green circle has the color value (R:0, G:127.5, B:100, A:204).

To apply our hypothetical color transformations to a real clip, we use a *Color* object as we saw earlier. To set a clip's universal color *offset* and *percentage* values we use the *setRGB( )* and *setTransform( )* methods (see the entries for those methods for example code). Conversely, to examine the current color transformations of a clip, we use the *getRGB( )* and *getTransform( )* methods. The *Color* class methods can produce animated color effects, such as fade-ins, fade-outs, and tinting. Furthermore, because we can apply tints to each clip instance individually, the *Color* class provides a very efficien way to create diverse graphics with minimal assets. For example, we can create a scene full of balloons from a single mov clip that we colorize and tint in myriad ways, as shown under the Example heading that follows.

Usage

Here are some points of interest for *Color* objects:

- Changing the color of a movie clip with a *Color* object will cause author-time tweens applied to the clip to cease functioning.
- Setting the `_alpha` property of a clip affects the clip's Alpha percentage, as reflected by the `aa` property of the object returned by *getTransform ( )* .
- Color transformations do not affect the background color of a movie or a movie clip. They apply only to solid shape placed on the Stage. To change the background color, first fill the background with a shape and then colorize that shape.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

- Manual color transformations can be applied to movie clips in the Flash MX authoring tool via the Property inspect and in Flash 5 via the Effect panel (Window Panels Effect). All such transformations are reflected in the properties of the object returned by *getTransform ( )* . The authoring tool serves as an excellent means of viewing a choosing color transformations.

Example

The first example shows how to generate a series of randomly colored balloon movie clips, based on an existing clip calle `balloon` :

```
var balloonColor;

// Loop to make 20 duplicates of the clip balloon
for (var i = 0; i < 20; i++) {
  // Duplicate this balloon
  balloon.duplicateMovieClip("balloon" + i, i);

  // Position this balloon on stage
  this["balloon" + i]._x = Math.floor(Math.random() * 550);
  this["balloon" + i]._y = Math.floor(Math.random() * 400);

  // Create a Color object for this balloon
  balloonColor = new Color(this["balloon" + i]);
  // Randomly assign this balloon's color using the setRGB() method
  balloonColor.setRGB(Math.floor(Math.random() * 0xFFFFFF));
}

// We don't need the Color object anymore, so delete it
delete balloonColor;
```

By setting the Red, Green, and Blue offsets to the same value, we can effectively brighten or darken a movie clip. For example, the following code darkens `myClip` :

```
brightness = new Color("myClip");
brightnessTransform = new Object();
brightnessTransform.rb = -30;
brightnessTransform.bb = -30;
brightnessTransform.gb = -30;
brightness.setTransform(brightnessTransform);
```

This last example adds a new method, *MovieClip.doMouseFade( )* , that brightens and darkens a clip according to the mou position:

```
MovieClip.prototype.doMouseFade = function () {
  var brightness = new Color(this);
  var brightnessTransform = new Object();

  this.onMouseMove = function () {
    var brightnessAmount = -255 + (_level0._xmouse / Stage.width) * 510;
    brightnessTransform.rb = brightnessAmount;
    brightnessTransform.bb = brightnessAmount;
    brightnessTransform.gb = brightnessAmount;
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

```
        brightness.setTransform(brightnessTransform);
        updateAfterEvent();
    }
}

box.doMouseFade();
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Color.getRGB( ) Method Flash 5

retrieve the current offset values for Red, Green, and Blue

```
colorObj.getRGB()
```

Returns

A number representing the current RGB offsets of *colorObj* 's target.

Description

The *getRGB( )* method returns a number, ranging from -16777215 to 16777215, that represents the current color offsets fo Red, Green, and Blue components in a clip; to retrieve the color percentages, you must use *getTransform( )* . Because colo values normally range from 0 to 255, it's convenient to work with the return value of *getRGB( )* in hexadecimal, where e offset can be represented by a two-digit hex number. To decipher the number returned by *getRGB( )* , we treat it as a six-d number of the form `0x RRGGBB` , where *RR* is the Red offset, *GG* is the Green offset, and *BB* is the Blue offset. For if *getRGB( )* returns the number 10092339, we convert it to the hex number `0x99FF33` , from which we derive the col (R:153, G:255, B:51). Or if *getRGB( )* returns the number 255, we convert it to the hex number `0x0000FF` , from whi derive the color offsets (R:0, G:0, B:255). The return value of *getRGB( )* can be converted to hexadecimal with *toString(* follows:

```
// Create a Color object
myColor = new Color("myClip");
// Set the Red offset to 255 (FF in hex)
myColor.setRGB(0xFF0000);
// Retrieve the RGB offset and convert it to hexadecimal
hexColor = myColor.getRGB().toString(16);
trace(hexColor);  // Displays: ff0000
```

Hexadecimal color values are familiar to most web developers, as they are often used in HTML tags. For example, here w hexadecimal number to specify the background color of an HTML page (equal values for red and blue combine to form pi

```
<BODY BGCOLOR="#FF00FF">
```

The hex color format used in HTML tags is, in fact, the same as the format used by *getRGB( )* and *setRGB( )* . However, mandatory to use hexadecimal to interpret the return value of *getRGB( )* ; we can also extract the individual Red, Green a color offsets from the return value of *getRGB( )* , using the bitwise operators:

```
var rgb = myColorObject.getRGB();
var red = (rgb >> 16) & 0xFF;   // Isolate the Red offset and assign it to r
var green = (rgb >> 8) & 0xFF;  // Isolate the Green offset and assign it to
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

```
var blue  = rgb & 0xFF;          // Isolate the Blue offset and assign it to
```

With the offset values separated into individual variables (one for each primary color), we can examine and manipulate th individually as decimal numbers. However, when we want to apply any offset value changes to a *Color* object, we must re the offsets into a single number, as shown in the entry for the *setRGB( )* method.

Usage

The *getRGB( )* and *setRGB( )* methods are convenient when we're assigning new colors directly to a clip, without referen clip's original color values. However, the *getTransform( )* and *setTransform( )* methods are better suited to modifying the l components of a clip's color transformation in relation to the clip's original colors.

Color offsets are most easily read using *getTransform( )* , which returns each component separately, rather than as an encc number containing all three colors as *getRGB( )* does. This is especially true when setting negative offsets with *setTransfc* due to the way that negative numbers are represented in binary.

Example

```
// Create a new Color object for a clip named box
boxColor = new Color("box");
// Set a new RGB offset for box
boxColor.setRGB(0x333366);
// Check the RGB offset for box
trace(boxColor.getRGB());      // Displays: 3355494
```

See Also

Color.getTransform( ) , Color.setRGB( )

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Chapter 18. ActionScript Language Reference

Color.getTransform( ) Method Flash 5

retrieve both the current offset and percentage values for a clip's Red, Green, Blue, and Alpha components

```
colorObj.getTransform()
```

Returns

An object whose properties contain the color transformation values for the target clip of `colorObj`.

Description

The *getTransform( )* method returns an object containing properties that tell us which transformations are currently applied to the target of a *Color* object. That is, it returns multiple individual values, wrapped in an object for convenience. The returned object is a instance of the generic *Object* class; it is not a *Color* object. The property names and values of the returned object are described in Table 18-5.

Table 18-5. Properties of object returned by getTransform( )

Property name	Property value	Property description
`ra`	-100 to 100	The Red transformation percentage
`rb`	-255 to 255	The Red offset amount
`ga`	-100 to 100	The Green transformation percentage
`gb`	-255 to 255	The Green offset amount
`ba`	-100 to 100	The Blue transformation percentage
`bb`	-255 to 255	The Blue offset amount
`aa`	-100 to 100	The Alpha transformation percentage
`ab`	-255 to 255	The Alpha offset amount

Usage

Note in Table 18-5 that both the percentage and the offset can have negative values; although negative percentages aren't meaningful in isolation, they can be factored in when calculating the RGB color components, which always range from 0 to 255. Values outside that range are clipped to the allowable range. See the *Color* class description for an explanation of the calculation used to determine final RGB and Alpha color components.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Example

We can use *getTransform( )* in combination with *setTransform( )* to modify the Red, Green, Blue, or Alpha components of a color transformation individually. For example, in the following code, we adjust the Red and Alpha components of a clip named `box`:

```
// Create a new Color object for a clip named box
boxColor = new Color("box");

// Assign the return object of getTransform( ) to boxTransform
boxTransform = boxColor.getTransform();

// Now, make some modifications to the transform object's properties
boxTransform.rb = 200;      // Set Red offset to 200
boxTransform.aa = 60;       // Set Alpha percentage to 60

// Apply the new transformations to box via boxColor
boxColor.setTransform(boxTransform);
```

See Also

Color.setTransform( )

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Color.setRGB( ) Method — Flash 5

assign new offset values for Red, Green, and Blue

```
colorObj.setRGB(offset);
```

Arguments

offset

A number in the range 0 to 16777215 (`0xFFFFFF`), representing the new RGB offsets of *`colorObj`*'s target clip. Can be a decimal integer or a hexadecimal integer. Numbers outside the allowed range are converted to numbers within the allowed range (using the rules of twos-complement binary notation). Therefore, *setRGB( )* cannot be used to set negative offset values (as *setTransform( )* can).

Description

The *setRGB( )* method assigns new transformation offsets for a movie clip's RGB components. The new *`offset`* is most easily specified as a six-digit hexadecimal number of the form `0x`*`RRGGBB`*, where *`RR`*, *`GG`*, and *`BB`* are two-digit numbers between 00 and FF, representing the Red, Green, and Blue components. For example, the RGB triplet (R:51, G:51, B:102) is equivalent to the hexadecimal value:

```
0x333366
```

Hence, to assign a gray RGB offset to a clip named `menu`, we can use:

```
var menuColor = new Color("menu");
menuColor.setRGB(0x999999);
```

Web developers comfortable with six-digit hexadecimal color values in HTML will have an easy time using *setRGB( )* with the preceding hexadecimal format. For a primer on decimal, octal, and hexadecimal numbers, see http://www.moock.org/asdg/technotes/basePrimer.

Note that in addition to setting offsets, *setRGB( )* also automatically sets the Red, Green, and Blue percentages of a clip's color transformation to 0, meaning that color changes performed via *setRGB( )* behave as direct color assignments (not adjustments of the original colors in the clip). To adjust the color of a movie clip in relation to the clip's original colors, use the *setTransform( )* method.

Example

Here's a handy technique for generating a number to use with the *setRGB( )* method. Our custom *combineRGB(*

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

) function shifts the `red` and `green` numbers into the proper position in a 24-bit number and then combines the `red`, `green`, and `blue` values using the bitwise *OR* operator (`|`). We use the result to assign a color value to the `box` movie clip. For more information on bitwise operations, see the online technote at http://www.moock.org/asdg/technotes/bitwise.

```
function combineRGB (red, green, blue) {
  // Combine the color values into a single number
  var RGB = (red<<16) | (green<<8) | blue;
  return RGB;
}
// Create the Color object
var boxColor = new Color("box");
// Set the color of box to the RGB triplet (R:201, G:160, B:21)
boxColor.setRGB(combineRGB(201, 160, 21));
```

The following example adds a convenient *setRGB( )* method to all movie clips. It automatically creates the necessary *Color* object and then calls the *Color.setRGB( )* method:

```
MovieClip.prototype.setRGB = function (colorValue) {
  new Color(this).setRGB(colorValue);
};

// Usage:
theClip_mc.setRGB(0xFF0000);  // Set theClip_mc's color to red
```

See Also

Color.getRGB( ), Color.setTransform( )

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Chapter 18. ActionScript Language Reference

Color.setTransform( ) Method — Flash 5

assign new offset and/or percentage values for Red, Green, Blue, and Alpha

```
colorObj.setTransform(transformObject)
```

Arguments

transformObject

An object whose properties contain the new color transformation values for the target clip of `colorObj` .

Description

The *setTransform( )* method gives us precise control over the percentage and offset of the Red, Green, Blue, and Alpha components of a movie clip's color. To use *setTransform( )* , we must first create an object (normally an instance of the *Ol* class) with a series of predefined properties. The transformation we wish to apply to our *Color* object is expressed using tl properties, which are listed in Table 18-5 for the *getTransform( )* method (except that *setTransform( )* sets their values rat than reading their values).

Once we have created an object with the properties described in Table 18-5, we pass that object to the *setTransform( )* method. The values of the properties on our `transformObject` become the new offset and percentage transform val of `colorObj` and, hence, `colorObj` 's target movie clip. The final color values in the clip are then determined according to the calculation discussed under the *Color* class Description.

To examine the current offsets and percentages of a particular *Color* object, we use the *getTransform( )* method.

Example

```
// Create a new Color object for the box clip
boxColor = new Color("box");

// Create a generic object (not a Color object) and store it in boxTransform
boxTransform = new Object();

// Assign the required properties of boxTransform (ra, rb, etc.), setting ou
// transformation values as desired
boxTransform.ra = 50;    // Red percentage
boxTransform.rb = 0;     // Red offset
boxTransform.ga = 100;   // Green percentage
boxTransform.gb = 25;    // Green offset
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

```
boxTransform.ba = 100;   // Blue percentage
boxTransform.bb = 0;     // Blue offset
boxTransform.aa = 40;    // Alpha percentage
boxTransform.ab = 0;     // Alpha offset

// Now that our transform object has been prepared, pass it to setTransform(
boxColor.setTransform(boxTransform);
```

This approach to creating a transform object is fairly labor-intensive. We can generate a new transform object more easily using the *getTransform( )* method, as follows:

```
// Create a new Color object for the box clip
boxColor = new Color("box");

// Invoke getTransform() on boxColor, which retrieves a transform object
boxTransform = boxColor.getTransform();

// Now alter boxTransform's properties as needed, leaving others unchanged
boxTransform.rb = 51;   // Red offset
boxTransform.aa = 40;   // Alpha percentage

// Use our transform object with setTransform( )
boxColor.setTransform(boxTransform);
```

See Also

Color.getTransform( )

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Chapter 18. ActionScript Language Reference

Date( ) Global Function — Flash 5

a string representing the current date and time

```
Date()
```

Returns

A string containing the current date and time.

Description

The *Date( )* function returns a human-readable string that expresses the current date and local time. The string also includes the GMT offset (the number of hours difference between local time and Greenwich Mean Time).

Be sure not to confuse the global *Date( )* function with the *Date( )* class constructor. The *Date( )* function returns the date formatted as a standard, if terse, string. It is convenient for humans but not very useful inside a program where you need to manipulate dates and times. For that purpose, you are better off using objects of the *Date* class, which allow convenient independent access to the year, month, day, and time.

Example

To place the current time and date in a text field with minimal fuss, we can use the *Date( )* function as follows:

```
someMovieClip.createTextField("output_txt", 1, 0, 0, 200, 20);
output_txt.text = Date();

// Sets the displayed text to a string formatted as follows:
// "Mon Aug 28 16:23:09 GMT-0400 2000"
```

Reader Exercise: Modify the preceding example using *String.substring( )* to show the date and time (including the hours and minutes), but eliminate the seconds and GMT offset from the display. See the Example in the *Date* class entry for a much more flexible approach.

See Also

The *Date* class, *Date.UTC( )*

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Chapter 18. ActionScript Language Reference

Date Class — Flash 5

current time and structured support for date information

Constructor

```
new Date()
new Date(milliseconds)
new Date(year, month, day, hours, minutes, seconds, ms)
```

Arguments

milliseconds

The number of milliseconds between the new date and midnight of January 1, 1970 UTC (Coordinated Universal T akin to GMT — positive if after; negative if before. Any required local time zone adjustment is made after the date UTC time is determined. For example, specifying a `milliseconds` argument of 1000 in Eastern Standard Tin creates a date 1 second past midnight on January 1, 1970 in UTC time, which translates to 7:00:01 P.M. on Decem 31, 1969 in EST time (5 hours behind UTC time).

year

An integer specifying the year. Required when using the `year , ...ms` constructor format. If `year` is one or tv digits, it is treated as the number of years since 1900 (e.g., a `year` of 11 refers to the year 1911, not 2011). Use digit numbers to specify year 2000 or later (e.g., use 2010, not 10). Three-digit years are treated as prior to 1000 A. In Flash 5, when `year` is negative or less than 800, the calculation is unreliable. In Flash 6, a negative year speci the number of years before 1900. To specify dates prior to 1000 A.D., it's safest to use the single `millisecon` constructor format.

month

An integer specifying the month, from 0 (January) to 11 (December), not from 1 to 12. Required when using the `year , ...ms` constructor format. Out-of-range months are carried over to the next or previous year. For exampl `month` argument of 13 is treated as February of the following year.

day

An optional integer specifying the day, from 1 to 31. Defaults to 1 if not specified. Out-of-range days are carried o to the next or previous month. For example, September 31 is treated as October 1, September 0 is treated as Augus and September -1 is treated as August 30.

hours

An optional integer specifying the hour, from 0 (midnight) to 23 (11 P.M.). Defaults to 0 if not specified. A.M. anc P.M. notation are not supported, so use 24-hour time. Out-of-range hours are carried over to the next or previous d

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

For example, an `hours` argument of 25 is treated as 1 A.M. of the following day.

minutes

An optional integer specifying the minute, from 0 to 59. Defaults to if not specified. Out-of-range minutes are carri over to the next or previous hour. For example, a `minutes` argument of 60 is treated as the first minute of the following hour.

seconds

An optional integer specifying the seconds, from 0 to 59. Defaults to if not specified. Out-of-range seconds are car over to the next or previous minute. For example, a `seconds` argument of -1 is treated as the last second of the previous minute.

ms

An optional integer specifying the milliseconds, from 0 to 999. Defaults to 0 if not specified. Out-of-range millisec are carried over to the next or previous second. For example, a `ms` argument of 1005 is treated as 5 milliseconds i the following second.

Class Methods

UTC( )

Retrieve the number of milliseconds between January 1, 1970 and a supplied UTC date.

Methods

getDate( )

Retrieve the day of the month, from 1 to 31.

getDay( )

Retrieve the day of the week, as a number from 0 (Sunday) to 6 (Saturday).

getFullYear( )

Retrieve the four-digit year.

getHours( )

Retrieve the hour of the specified *Date* object, from 0 to 23.

getMilliseconds( )

Retrieve the milliseconds of the specified *Date* object, from 0 to 999.

getMinutes( )

Retrieve the minutes of the specified *Date* object, from 0 to 59.

getMonth( )

Retrieve the month of the year, as a number from 0 (January) to 11 (December).

getSeconds( )

Retrieve the seconds of the specified *Date* object, from 0 to 59.

getTime( )

Retrieve the date in internal format (i.e., the number of milliseconds between the date and January 1, 1970).

getTimezoneOffset( )

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Retrieve the number of minutes between UTC and local time.

getUTCDate( )

Retrieve the day of the month, from 1 to 31, in UTC time.

getUTCDay( )

Retrieve the day of the week, from 0 (Sunday) to 6 (Saturday), in UTC time.

getUTCFullYear( )

Retrieve the four-digit year in UTC time.

getUTCHours( )

Retrieve the hour of the day, from 0 to 23, in UTC time.

getUTCMilliseconds( )

Retrieve the milliseconds, from 0 to 999, in UTC time.

getUTCMinutes( )

Retrieve the minutes, from 0 to 59, in UTC time.

getUTCMonth( )

Retrieve the month of the year, from 0 (January) to 11 (December), in UTC time.

getUTCSeconds( )

Retrieve the seconds, from 0 to 59, in UTC time.

getYear( )

Retrieve the year, relative to 1900.

setDate( )

Assign the day of the month.

setFullYear( )

Assign the year in four-digit format.

setHours( )

Assign the hour of the day.

setMilliseconds( )

Assign the milliseconds.

setMinutes( )

Assign the minutes.

setMonth( )

Assign the month of the year.

setSeconds( )

Assign the seconds.

setTime( )

Assign the date in internal format (i.e., the number of milliseconds between the date and January 1, 1970).

setUTCDate( )

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Assign the day of the month in UTC time.

setUTCFullYear( )

Assign the four-digit year in UTC time.

setUTCHours( )

Assign the hour of the day in UTC time.

setUTCMilliseconds( )

Assign the milliseconds in UTC time.

setUTCMinutes( )

Assign the minutes in UTC time.

setUTCMonth( )

Assign the month of the year in UTC time.

setUTCSeconds( )

Assign the seconds in UTC time.

setYear( )

Assign the year in four-digit format, or in two-digit format for the twentieth century.

toString( )

A human-readable string representing the date.

valueOf( )

The number of milliseconds between the date and midnight of January 1, 1970 UTC.

Description

We use objects of the *Date* class to determine the current time and date and as a means of storing arbitrary dates and times structured format.

In ActionScript, a specific date is represented by the number of milliseconds before or after midnight of January 1, 1970. I the number of milliseconds is positive, the date comes after midnight, January 1, 1970; if the number of milliseconds is negative, the date comes before midnight, January 1, 1970. For example, if a date value is 10000, the date being described 12:00:10, January 1, 1970. If a date value is -10000, the date being described is 11:59:50, December 31, 1969.

Normally, however, we needn't worry about calculating the number of milliseconds to a particular date; the ActionScript interpreter takes care of that for us. When we construct a *Date* object, we simply describe our date as a year, month, day, l minute, second, and millisecond. The interpreter then converts that point in time to the internal milliseconds-from-1970 format. We can also ask the interpreter to create a new *Date* object using the current time. Once a *Date* object is created, v use the *Date* class's methods to retrieve and set the date's year, month, day, hour, minute, second, and millisecond.

There are three ways to make a new *Date* object:

- Invoke the *Date ( )* constructor with no arguments. This sets the new *Date* object to the current time.
- Iinvoke the *Date ( )* constructor with one numeric argument: the number of milliseconds between midnight, January 1970 and the date we're creating.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

- Invoke the *Date ( )* constructor with two to seven numeric arguments, corresponding to the year and month (mandat and (optionally) the day, hour, minute, second, and millisecond of the date we're creating.

Because dates are stored internally as a single number, objects of the *Date* class do not have properties to retrieve and set. Rather, they have methods that we use to access the various components of a date in human-readable terms (i.e., in more convenient units). For example, to determine the month of a particular *Date* object, we use:

```
myMonth = myDate.getMonth();
```

We cannot use:

```
myMonth = myDate.month;  // There's no such property! We have to use methods
```

Many *Date* methods contain the letters "UTC," which is an abbreviation for Coordinated Universal Time. For most purpos UTC time is directly synonymous with the more colloquial Greenwich Mean Time, or GMT, the time as measured on the Greenwich meridian. UTC simply is a newer standard that is less ambiguous than GMT, which has had several meanings history. The UTC methods allow us to work with times directly in Coordinated Universal Time without converting betwe time zones. The equivalent non-UTC methods all generate values based on local, adjusted time zones.

Usage

All dates and times, including the UTC times, are determined according to the settings on the client-side operating system which the Flash movie is running (not the web server) and include regional offsets and, in some cases, daylight savings tir Times and dates, therefore, are only as accurate as the user's system.

When *Date( )* is used as a global function — without the `new` keyword — it generates a string that expresses the current t the same format as would be returned by *Date.toString( )* .

ActionScript does not include a built-in means to convert a string into a *Date* object, as is supported in JavaScript.

Example

Dates can be added and subtracted to calculate cumulative times or elapsed times. Suppose our friend Graham decides to g traveling for a little less than a year. While he's gone, we want to keep track of how many days he's been away and how m days we have to wait for him to return. The following code takes care of our countdown:

```
// Assign the current time to now.
var now = new Date();

// Graham leaves September 7, 2000 (remember, months start at 0,
// so September is 8, not 9).
var departs = new Date(2000, 8, 7);

// Graham comes back August 15, 2001.
var returns = new Date(2001, 7, 15);

// Convert the times to milliseconds for easy comparison.
// Then check how many milliseconds have elapsed between the two times.
var gone = now.getTime() - departs.getTime();

// Divide the difference between the departure date and now by the number
// of milliseconds in a day. This tells us how many days we've been waiting.
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

```
var numDaysGone = Math.floor(gone/86400000);

// Use the same technique to determine how many days we have left to wait.
// But this time round up (using the ceil( ) function) instead of rounding d
var left = returns.getTime() - now.getTime();
var numDaysLeft = Math.ceil(left/86400000);

// Display our days in text fields.
this.createTextField("goneOutput_txt", 1, 0, 0, 200, 20);
this.createTextField("leftOutput_txt", 2, 0, 40, 200, 20);
goneOutput_txt.text = "gone for: " + numDaysGone + " days";
leftOutput_txt.text = "returns in: " + numDaysLeft + " days";
```

When adding a day to, or subtracting a day from, an existing date, we normally assign the number of milliseconds in a day (86400000) to a variable, for convenient use. The following code adds one day to the current date:

```
oneDay = 86400000;
now = new Date();
tomorrow = new Date(now.getTime() + oneDay);

// We could also add one day to the date in now like this
now.setTime(now.getTime() + oneDay);
```

To apply custom formatting to a date, we must manually map the return values of *getDate( )* , *getDay( )* , *getHours( )* , and on, to custom strings as shown in the following example:

```
// Takes any Date object and returns a string of the format:
// Saturday December 16
function formatDate (theDate) {
  var months = ["January", "February", "March", "April",
                "May", "June", "July", "August", "September",
                "October", "November", "December"];
  var days = ["Sunday", "Monday", "Tuesday", "Wednesday", "Thursday",
               "Friday", "Saturday"];
  var dateString = days[theDate.getDay()] + " "
                   + months[theDate.getMonth()] + " "
                   + theDate.getDate();
  return dateString;
}

now = new Date();
trace("Today is " + formatDate(now));
```

The next example shows how to convert the 24-hour clock return value of *getHours( )* to a 12-hour clock with AM and P

```
// Takes any Date object and returns a string of the format:
// "2:04PM"
function formatTime (theDate) {
  var hour = theDate.getHours();
  var minute = theDate.getMinutes() > 9 ?
               theDate.getMinutes() : "0" + theDate.getMinutes();
  if (hour > 12) {
    var timeString = (hour - 12) + ":" + minute + "PM";
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

```
  } else {
    var timeString = hour + ":" + minute + "AM";
  }
  return timeString;
}

now = new Date();
trace("The time is " + formatTime(now));
```

For an example that uses the *Date* class to create an analog-style clock, see Section 13.11 in Chapter 13. Note that programming time-based behavior, such as a 10-second pause in a movie, is often accomplished more easily with the glob functions *getTimer( )* and *setInterval( )* .

See Also

Date( ) , *Date.UTC( )* , *getTimer( )*

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Chapter 18. ActionScript Language Reference

Date.getDate( ) Method Flash 5

the day of the month

```
date.getDate()
```

Returns

An integer from 1 to 31, representing the day of the month for `date`.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Chapter 18. ActionScript Language Reference

Date.getDay( ) Method Flash 5

the day of the week

```
date.getDay()
```

Returns

An integer from 0 (Sunday) to 6 (Saturday), representing the day of the week for `date`.

Example

The following code loads a *.swf* file specific to the current day of the week into the movie clip `welcomeHeader` (the seven *.swf* files are named in the series *sun.swf*, *mon.swf*, and so on):

```
now = new Date();
today = now.getDay();
days = ["sun", "mon", "tue", "wed", "thu", "fri", "sat"];
welcomeHeader.loadMovie(days[today] + ".swf");
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Chapter 18. ActionScript Language Reference

Date.getFullYear( ) Method Flash 5

the four-digit year

```
date.getFullYear()
```

Returns

A four-digit integer, representing the year for `date` (for example, 1999 or 2000).

See Also

Date.getYear( )

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Chapter 18. ActionScript Language Reference

Date.getHours( ) Method Flash 5

the hour of the day

```
date.getHours()
```

Returns

An integer from 0 (midnight) to 23 (11 P.M.), representing the hour of the day for `date`. A.M. and P.M. notation are not supported but can be constructed manually, as shown in the *Date* class examples.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Chapter 18. ActionScript Language Reference

Date.getMilliseconds( ) Method Flash 5

the milliseconds of a date

```
date.getMilliseconds()
```

Returns

An integer from 0 to 999, representing the milliseconds of `date`. Note that it does not represent the milliseconds from 1970 — see *Date.getTime( )* — rather, it represents the fractional remainder of the seconds indicated by the specified *Date* object.

See Also

Date.getTime( )

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Chapter 18. ActionScript Language Reference

Date.getMinutes( ) Method Flash 5

the minutes of a date

```
date.getMinutes()
```

Returns

An integer from 0 to 59, representing the minutes of the hour of `date`.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Date.getMonth( ) Method Flash 5

the month of the year

```
date.getMonth()
```

Returns

An integer from 0 (January) to 11 (December), not 1 to 12, representing the month of the year of `date`.

Usage

Be careful not to assume that 1 is January! The return value of *getMonth( )* starts at 0, not 1.

Example

Here we convert the number returned by *getMonth( )* to a human-readable abbreviation:

```
var months = ["Jan", "Feb", "Mar", "Apr", "May", "Jun",
              "Jul", "Aug", "Sept", "Oct", "Nov", "Dec"];
myDateObj = new Date();
trace("The month is " + months[myDateObj.getMonth()]);
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Chapter 18. ActionScript Language Reference

Date.getSeconds( ) Method Flash 5

the seconds of a date

```
date.getSeconds()
```

Returns

An integer from 0 to 59, representing the seconds of the minute of `date`.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Chapter 18. ActionScript Language Reference

Date.getTime( ) Method — Flash 5

retrieve the number of milliseconds between January 1, 1970 and the time of a Date object

```
date.getTime()
```

Returns

An integer expressing the number of milliseconds between the time of `date` and midnight, January 1, 1970 — positive if after January 1, 1970; negative if before.

Description

Internally, all dates are represented as a single number—the number of milliseconds between the time of the `date` and midnight, January 1, 1970. The *getTime( )* method gives us access to that number of milliseconds so that we can use it to construct other dates or calculate the elapsed time between two dates.

Example

Suppose we place the following code on frame 10 of a movie:

```
time1 = new Date();
```

Then we place the following code on frame 20 of a movie:

```
time2 = new Date();
```

With the following code, we can determine the amount of time that elapsed, in milliseconds, between frames 10 and 20 of our movie:

```
elapsedTime = time2.getTime() - time1.getTime();
```

Note that Flash's global *getTimer( )* function also gives us access to elapsed time in a movie.

See Also

Date.setTime( ), getTimer( ), setInterval( )

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Chapter 18. ActionScript Language Reference

Date.getTimezoneOffset( ) Method Flash 5

the number of minutes between local time and UTC (a.k.a. GMT)

```
date.getTimezoneOffset()
```

Returns

An integer representing the current number of *minutes* between the local time zone and the actual UTC (Greenwich meridian) time — positive if local time is behind UTC; negative if local time is ahead of UTC. Includes adjustments for local daylight savings time depending on the day of the year.

Example

When invoked in Eastern Daylight Time (EDT) during daylight savings, the following code returns the value 240 (240 minutes is 4 hours):

```
myDate = new Date();
trace(myDate.getTimezoneOffset());  // Displays: 240
```

However, when invoked in EDT, during non-daylight savings times of year, the same code returns 300 (300 minutes is 5 hours), which is the offset between Eastern Standard Time (EST) and UTC.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Chapter 18. ActionScript Language Reference

Date.getUTCDate( ) Method Flash 5

the day of the month (UTC time)

```
date.getUTCDate()
```

Returns

An integer from 1 to 31, representing the day of the month for `date`, where `date` is treated as a UTC time.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Chapter 18. ActionScript Language Reference

Date.getUTCDay( ) Method Flash 5

the day of the week (UTC time)

```
date.getUTCDay()
```

Returns

An integer from 0 (Sunday) to 6 (Saturday), representing the day of the week for `date`, where `date` is treated as a UTC time.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Chapter 18. ActionScript Language Reference

Date.getUTCFullYear( ) Method Flash 5

the four-digit year (UTC time)

```
date.getUTCFullYear()
```

Returns

A four-digit integer, representing the year for *date*, where *date* is treated as a UTC time (for example, 1999 or 2000).

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Chapter 18. ActionScript Language Reference

Date.getUTCHours( ) Method Flash 5

the hour of the day (UTC time)

```
date.getUTCHours()
```

Returns

An integer from 0 (midnight) to 23 (11 P.M.), representing the hour of the day for `date`, where `date` is treated as a UTC time.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Chapter 18. ActionScript Language Reference

Date.getUTCMilliseconds( ) Method Flash 5

the milliseconds of a date (UTC time)

```
date.getUTCMilliseconds()
```

Returns

An integer from 0 to 999, representing the milliseconds of `date`, where `date` is treated as a UTC time.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Chapter 18. ActionScript Language Reference

Date.getUTCMinutes( ) Method Flash 5

the minutes of a date (UTC time)

```
date.getUTCMinutes()
```

Returns

An integer from 0 to 59, representing the minutes of the hour of `date`, where `date` is treated as a UTC time.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Chapter 18. ActionScript Language Reference

Date.getUTCMonth( ) Method Flash 5

the month of the year (UTC time)

```
date.getUTCMonth()
```

Returns

An integer from 0 (January) to 11 (December), not 1 to 12, representing the month of the year of `date`, where `date` is treated as a UTC time.

Usage

Be careful not to assume that 1 is January! The return value of *getUTCMonth( )* starts at 0, not 1.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Chapter 18. ActionScript Language Reference

Date.getUTCSeconds( ) Method Flash 5

the seconds of a date (UTC time)

```
date.getUTCSeconds()
```

Returns

An integer from 0 to 59, representing the seconds of the minute of `date`, where `date` is treated as a UTC time.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Chapter 18. ActionScript Language Reference

Date.getYear( ) Method Flash 5

the year, relative to 1900

```
date.getYear()
```

Returns

The value of *date*.*getFullYear( )* minus 1900. For example, *getYear( )* of 1999 is 99, *getYear( )* of 2001 is 101, and *getYear( )* of 1800 is -100. This function is most useful for dates in the twentieth century.

See Also

Date.getFullYear( )

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Date.setDate( ) Method Flash 5

assign the day of the month

```
date.setDate(day)
```

Arguments

day

An integer from 1 to 31, representing the new day of the month for `date`. If you specify a day greater than the number of days in the current month, it causes the month to increment accordingly. For example, if the current `month` is 8 (September), and you specify 31 for the new `day`, the date is treated as October 1. The `day` becomes 1, and the `month` becomes 9 (October).

Returns

An integer representing the number of milliseconds between the new date and midnight, January 1, 1970.

See Also

Date.getDate( ), Date.setMonth( )

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Chapter 18. ActionScript Language Reference

Date.setFullYear( ) Method Flash 5

assign the century and year in four-digit format

```
date.setFullYear(year, month, day)
```

Arguments

year

A four-digit integer, representing the new year for `date` (for example, 1999 or 2000).

month

An optional integer from 0 (January) to 11 (December), not 1 to 12, representing the new month of the year of `date`. Defaults to 0 if not specified.

day

An optional integer from 1 to 31, representing the new day of the month of `date`. Defaults to 1 if not specified.

Returns

An integer representing the number of milliseconds between the new date and midnight, January 1, 1970.

See Also

Date.getFullYear( ), Date.setYear( )

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Chapter 18. ActionScript Language Reference

Date.setHours( ) Method Flash 5

assign the hour of the day

```
date.setHours(hour)
```

Arguments

hour

An integer from 0 (midnight) to 23 (11 P.M.), specifying the new hour of the day for `date`.

Returns

An integer representing the number of milliseconds between the new date and midnight, January 1, 1970.

See Also

Date.getHours( )

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Chapter 18. ActionScript Language Reference

Date.setMilliseconds( ) Method Flash 5

assign the milliseconds of a date

```
date.setMilliseconds(ms)
```

Arguments

ms

An integer from 0 to 999, representing the new milliseconds of `date`, not the number of milliseconds since 1970. Values above 999 or below 0 are carried over to the seconds of `date`.

Returns

An integer representing the number of milliseconds between the new date and midnight, January 1, 1970.

See Also

Date.getMilliseconds( ), Date.setTime( )

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Date.setMinutes( ) Method Flash 5

assign the minutes of a date

```
date.setMinutes(minutes)
```

Arguments

minutes

An integer from 0 to 59, representing the new minutes of the hour of `date`. Values above 59 or below 0 are carried over to the hours of `date`.

Returns

An integer representing the number of milliseconds between the new date and midnight, January 1, 1970.

See Also

Date.getMinutes( )

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Date.setMonth( ) Method Flash 5

assign the month of the year

```
date.setMonth(month)
```

Arguments

month

An integer from 0 (January) to 11 (December), not from 1 to 12, representing the new month of the year of `date`. Values above 11 or below 0 are carried over to the year of `date`.

Returns

An integer representing the number of milliseconds between the new date and midnight, January 1, 1970.

Usage

Be careful not to assume that 1 is January! Months start at 0, not 1.

See Also

Date.getMonth( )

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Chapter 18. ActionScript Language Reference

Date.setSeconds( ) Method Flash 5

assign the seconds of a date

```
date.setSeconds(seconds)
```

Arguments

seconds

An integer from 0 to 59, representing the new seconds of the minute of `date`. Values above 59 or below 0 are carried over to the minutes of `date`.

Returns

An integer representing the number of milliseconds between the new date and midnight, January 1, 1970.

See Also

Date.getSeconds( )

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Chapter 18. ActionScript Language Reference

Date.setTime( ) Method — Flash 5

assign a new date based on the number of milliseconds between January 1, 1970 and the new date

date.setTime(*milliseconds*)

Arguments

milliseconds

An integer expressing the number of milliseconds between the new desired date and midnight, January 1, 1970 — positive if after January 1, 1970; negative if before.

Returns

The value of *milliseconds*.

Description

Internally, all dates are represented as the number of milliseconds between the time of the date and midnight, January 1, 1970. The *setTime( )* method specifies a new date using the internal millisecond representation. Setting a date using milliseconds from 1970 is often handy when we're determining differences between multiple dates and times using *getTime( ).*

Example

Using *setTime( )* in concert with *getTime( )*, we can adjust the time of an existing date by adding or subtracting milliseconds. For example, here we add one hour to a date:

```
now = new Date();
now.setTime(now.getTime() + 3600000);
```

And here we add one day:

```
now = new Date();
now.setTime(now.getTime() + 86400000);
```

To improve the readability of our code, we create variables representing the number of milliseconds in an hour and milliseconds in a day:

```
oneDay = 86400000;
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

```
oneHour = 3600000;
now = new Date();
// Subtract one day and three hours.
now.setTime(now.getTime() - oneDay - (3 * oneHour));
```

See Also

Date.getTime( ), Date.setMilliseconds( ), Date.UTC( ), getTimer( )

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Chapter 18. ActionScript Language Reference

Date.setUTCDate( ) Method — Flash 5

assign the day of the month (UTC time)

```
date.setUTCDate(day)
```

Arguments

day

An integer from 1 to 31, representing the new day of the month for `date` in UTC time. If you specify a day greater than the number of days in the current month, it causes the month to increment accordingly. For example, if the current `month` is 8 (September) and you specify 31 for the new `day`, it is treated as October 1. The `day` becomes 1, and the `month` becomes 9 (October).

Returns

An integer representing the number of milliseconds between the new date and midnight, January 1, 1970, in UTC time.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Chapter 18. ActionScript Language Reference

Date.setUTCFullYear( ) Method Flash 5

assign the century and year in four-digit format (UTC time)

```
date.setUTCFullYear(year)
```

Arguments

year

A four-digit integer, representing the new year for `date`, in UTC time (for example, 1999 or 2000).

Returns

An integer representing the number of milliseconds between the new date and midnight, January 1, 1970, in UTC time.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Date.setUTCHours( ) Method Flash 5

assign the hour of the day (UTC time)

```
date.setUTCHours(hour)
```

Arguments

hour

An integer from 0 (midnight) to 23 (11 P.M.), representing the new hour of the day for `date`, in UTC time.

Returns

An integer representing the number of milliseconds between the new date and midnight, January 1, 1970, in UTC time.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Chapter 18. ActionScript Language Reference

Date.setUTCMilliseconds( ) Method Flash 5

assign the milliseconds of a date (UTC time)

```
date.setUTCMilliseconds(ms)
```

Arguments

ms

An integer from 0 to 999, representing the new milliseconds of `date`, in UTC time.

Returns

An integer representing the number of milliseconds between the new date and midnight, January 1, 1970 in UTC time.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Chapter 18. ActionScript Language Reference

Date.setUTCMinutes( ) Method Flash 5

assign the minutes of a date (UTC time)

```
date.setUTCMinutes(minutes)
```

Arguments

minutes

An integer from 0 to 59, representing the new minutes of the hour of `date`, in UTC time.

Returns

An integer representing the number of milliseconds between the new date and midnight, January 1, 1970, in UTC time.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Date.setUTCMonth( ) Method Flash 5

assign the month of the year (UTC time)

```
date.setUTCMonth(month)
```

Arguments

month

An integer from 0 (January) to 11 (December), not from 1 to 12, representing the new month of the year of `date`, in UTC time.

Returns

An integer representing the number of milliseconds between the new date and midnight, January 1, 1970, in UTC time.

Usage

Be careful not to assume that 1 is January! Months start at 0, not 1.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Chapter 18. ActionScript Language Reference

Date.setUTCSeconds( ) Method Flash 5

assign the seconds of a date (UTC time)

```
date.setUTCSeconds(seconds)
```

Arguments

seconds

An integer from 0 to 59, representing the new seconds of the minute of `date`, in UTC time.

Returns

An integer representing the number of milliseconds between the new date and midnight, January 1, 1970, in UTC time.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Chapter 18. ActionScript Language Reference

Date.setYear( ) Method Flash 5

assign the year, relative to 1900

```
date.setYear(year, month, day)
```

Arguments

year

A required integer specifying the new year of `date`. If one or two digits are supplied, the year is assumed to be in the twentieth century (1900s). For example, 1 is the year 1901, and 99 is the year 1999. Three-digit years are assumed to be prior to 1000 A.D. Use four digits to specify the year 2000 and later. A negative number specifies the number of years B.C., not the number of years before 1900.

month

An optional integer from 0 (January) to 11 (December), not from 1 to 12, representing the new month of the year of `date`.

day

An optional integer from 1 to 31, representing the new day of the month of `date`.

Returns

An integer representing the number of milliseconds between the new date and midnight, January 1, 1970.

Description

setYear( ) is identical to *setFullYear( )*, except that it interprets one- and two-digit years as being relative to 1900, whereas *setFullYear( )* interprets them as being relative to 0 A.D.

See Also

Date.getYear( ), Date.setFullYear( )

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Chapter 18. ActionScript Language Reference

Date.toString( ) Method Flash 5

a human-readable string representing the date

```
date.toString()
```

Returns

A string giving the time and date of *date* in human-readable format, including the UTC (GMT) offset. To compose customized representations of the date, use the methods for retrieving the day, hours, minutes, and so on, and concatenate those values to form your own strings, as shown earlier in the Example under *Date* class.

Example

```
trace (myDate.toString()); // Displays a date in the format:
                           // Wed Sep 15 12:11:33 GMT-0400 1999
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Chapter 18. ActionScript Language Reference

Date.UTC( ) Class Method — Flash 5

retrieve the number of milliseconds between January 1, 1970 and a supplied UTC date

```
date.UTC(year, month, day, hours, minutes, seconds, ms)
```

Arguments

year, ...ms

A series of numeric values describing the date and time, supplied in UTC time, not local time. For descriptions of each argument see the *Date( )* constructor.

Returns

The number of milliseconds between the specified date and midnight, January 1, 1970.

Description

The *Date.UTC( )* method takes the same arguments as the *Date( )* constructor, but instead of returning an object for the specified date, *Date.UTC( )* returns a number indicating the date in the internal milliseconds-from-1970 format. The returned number is typically used to construct a new *Date* object in UTC time or to assign a UTC time to an existing *Date* object via the *setTime( )* method.

Example

The following code shows how to measure the milliseconds elapsed between midnight, January 1, 1970 and midnight, January 1, 2000 in UTC time:

```
trace(Date.UTC(2000, 0) + " milliseconds passed between 1970 and 2000.");
// Displays: "946684800000 milliseconds passed between 1970 and 2000."
```

Here we use those elapsed milliseconds to construct a UTC-time-based *Date* object:

```
nowUTC = new Date(Date.UTC(2000, 0));
```

If we invoke that code in EST (Eastern Standard Time), which is 5 hours behind UTC, `nowUTC` represents the local time, 7 P.M. on December 31, 1999. When we check the hour using the non-UTC method *getHours( )*, we get the local hour, 19 (7 P.M. in a 24-hour clock):

```
trace(nowUTC.getHours());  // Displays: 19
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

But when we check the hour using the UTC method *getUTCHours( )*, we get the correct UTC hour, (midnight in a 24-hour clock):

```
trace(nowUTC.getUTCHours());  // Displays: 0
```

See Also

Date( ), *Date.setTime( )*, the *Date* class

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Chapter 18. ActionScript Language Reference

Date.valueOf( ) Method — Flash 5

the number of milliseconds between January 1, 1970 and the time of the Date object

```
date.valueOf()
```

Returns

An integer expressing the number of milliseconds between the time of the *Date* object and midnight, January 1, 1970 — positive if after January 1, 1970; negative if before.

Description

In practice, *Date.valueOf( )* is equivalent to *Date.getTime( )*.

See Also

Date.toString( )

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Chapter 18. ActionScript Language Reference

delete Operator Flash 5

delete the contents of a variable, property, or array element

```
delete identifier
```

Operand

identifier

The identifier to be deleted.

Returns

The Boolean `true` if the *`identifier`* was deleted, `false` if not (because the *`identifier`* doesn't exist or its value cannot be deleted).

Description

The *delete* operator removes an object, an object property, an array element, or variable from a script.

See Also

"The *delete* Operator," in Chapter 5

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

duplicateMovieClip( ) Global Function Flash 4

create a copy of a movie clip

```
duplicateMovieClip(target, newName, depth)
```

Arguments

target

A string indicating the path to the movie clip to duplicate (known as the *seed clip*). Nested clips can be referred to with dot syntax, as in *duplicateMovieClip ("_root.myClip", "myClip2", 0)*. Because a movie clip reference is converted to a path when used in a string context, `target` can also be a movie clip object reference, as in *duplicateMovieClip(_root.myClip, "myClip2", 0)*.

newName

A string that becomes the instance name of the duplicated clip. The string used must adhere to the rules for creating an identifier outlined in Section 15.5 in Chapter 15.

depth

An integer, between -16384 and 1048575 (inclusive), specifying the depth on which to place the duplicated clip inside `target`'s parent. Higher depths obscure lower depths, so a depth of 6 is in front of 5, which is in front of 4, and so on. Depths above -1 are reserved for dynamically generated content. Therefore, in most cases, you should use a depth between 0 and 1048575 with *duplicateMovieClip( )*. However, when creating dynamic content that you want to appear beneath all author-time content in `target` (e.g., a run-time background), use depth -16384. Depths from -16383 to -1 (inclusive) are reserved for author-time content and shouldn't be used with *duplicateMovieClip( )*. For important details, see Section 13.4.

Description

The global *duplicateMovieClip( )* function is one way to create a new movie clip during movie playback—others are *MovieClip.attachMovie( )*, *MovieClip.createEmptyMovieClip( )*, and *MovieClip.duplicateMovieClip( )*. The *duplicateMovieClip( )* function creates an identical copy of `target` and places the copy in `target`'s parent's clip stack on layer `depth`. The duplicated clip begins playing at frame 1, no matter which frame `target` is at when duplication occurs.

A duplicated clip inherits any transformations (rotation, scale, etc.) that had been applied to `target`, but it does not inherit the timeline variables of `target`. Callback functions assigned to `target`'s event handler properties (e.g., *onEnterFrame( )*) are not copied to the duplicated clip. However, Flash 5-style *onClipEvent( )* handlers are copied to the duplicate. The global *duplicateMovieClip( )* function is also available as a movie clip method, though when used in that form, the `target` argument is not used.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

To delete a duplicated clip, use *removeMovieClip( )* (see the global *removeMovieClip( )* function description for more important details).

Bugs

The global form of *duplicateMovieClip( )* lacks the following abilities added to *MovieClip.duplicateMovieClip( )* in Flash 6:

- The global form does not include support for the `initObject` parameter (used to initialize properties of the newly created clip).
- The global form does not return a reference to the duplicated clip.

Example

```
// Copy the ball_mc clip and name the copy ball2_mc
duplicateMovieClip(ball_mc, "ball2_mc", 0);
// Move the new ball2_mc clip over so we can see it
ball2_mc._x += 100;
```

See Also

MovieClip.duplicateMovieClip( ), *removeMovieClip( )*; "Creating instances with duplicateMovieClip( )," "How clips generated via duplicateMovieClip( ) are added to the stack," and "Method versus global function overlap issues," in Chapter 13

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Chapter 18. ActionScript Language Reference

#endinitclip Pragma — Flash 6

ends a movie clip initialization code block

```
#endinitclip
```

Description

The `#endinitclip` pragma ends a special block of code (begun with `#initclip`) that initializes a movie clip symbol before any instances of it are created. It can appear only on frame 1 of a movie clip symbol (it causes a compile-time error elsewhere) and must be preceded by `#initclip`. For complete details, see `#initclip`.

See Also

`#initclip`

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Chapter 18. ActionScript Language Reference

escape( ) Global Function — Flash 6

encode a string for safe network transferFlash 5; enhanced to support UTF-8 in

`escape(`*`string`*`)`

Arguments

string

A string (or an expression that yields a string) to be encoded.

Returns

An encoded version of *`string`* .

Description

The *escape ( )* function creates a new encoded string, based on a supplied string. The new string contains a hexadecimal es sequence in place of any character in the supplied string that is not a digit or a basic, unaccented Latin letter between A an a and z. As of Flash Player 6, the replacement hexadecimal escape sequences take the following format:

- If `System.useCodepage` is `false` (the default), the character is replaced by one or more hexadecimal escap sequences of the form `%xx` , representing the UTF-8 sequence for the character. Characters in the Unicode range 0- encoded as `%xx` (single-byte), characters 128-2047 are encoded as `%xx%xx` (double-byte), and characters 2048 an are encoded as `%xx%xx%xx` (triple-byte).
- If `System.useCodepage` is `true` and the system is not an Asian language double-byte system, the character replaced with a hexadecimal escape sequence `%xx` , where `xx` is the hex code point for the character. The code poi determined using the system code page (typically Latin 1 on most Western computers). Note that on Macs, the syste code page is Mac Roman, but Flash uses Latin 1 instead.
- If `System.useCodepage` is `true` and the system is an Asian language double-byte system, the character is re with a hexadecimal escape sequence `%xx` or `%xx%xx` , using the double-byte character encoding supported by the (for example, Shift-JIS).

Flash Player 5 behaves as if `System.useCodepage` is always `true` (earlier versions did not support *escape( )*).

The *escape ( )* function is sometimes used when a Flash movie sends information to server applications or writes cookies i browser. To decode an encoded string, use the global *unescape ( )* function.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Usage

Note that for characters above Unicode code point U+007F (decimal 127), the UTF-8 escape sequence is not the same nur the Unicode code point! For example, the code point for e is U+00E9 (decimal 233), but the UTF-8 sequence is %C3%A9 the *LoadVars* class for information on loading external encoded text.

Example

```
trace(escape("6-5 is 4"));  // Displays: 6%2D5%20is%204  (- is %2D, space is
trace(escape("resume"));    // Displays: resum%C3%A9
```

See Also

unescape( ) ; Appendix B

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Chapter 18. ActionScript Language Reference

eval( ) Global Function — Flash 6

interpret a string as an identifierFlash 4; illegal on left side of assignment in

`eval(`*`stringExpression`*`)`

Arguments

stringExpression

A string expression that matches some identifier. Can be qualified (e.g., "_root.speed") or unqualified (e.g., "speed

Returns

The value of the variable represented by *`stringExpression`* or a reference to the object, movie clip, or function represented by *`stringExpression`* . If *`stringExpression`* does not represent an existing identifier, `undefined` is returned.

Description

The *eval ( )* function provides a means of dynamically constructing an identifier based on a string of text. It looks up the variable identified by *`stringExpression`* and returns its value. For example, here we use the return value of *eval (* set the value of `currentName` :

```
var name1 = "Kathy";
var count = 1;
// Look up name1 and return its value, "Kathy", which we assign to currentNa
var currentName = eval("name" + count);
```

And here we control a dynamically named movie clip, `star1` :

```
var count = 1;
eval("star" + count)._x += 100;        // Move star1 right 100 pixels
```

In Flash 4 and 5 (but not Flash 6), we can also use an *eval ( )* invocation in place of an identifier that is the left-hand oper of an assignment expression, as in:

```
var count = 4;
eval("name" + count) = "Simone";      // Sets name4 to "Simone" in Flash 4 an
```

However, in Flash 6 *eval ( )* cannot be used on the left side of an assignment (in keeping with the ECMA-262 specificatio

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

To dynamically construct the left-hand side of an assignment expression, use the property access operator, `[ ]` , or the *se* statement instead:

```
// Using [  ]
var count = 4;
this["name" + count] = "Simone";    // Sets name4 to "Simone"
trace (name4);

// Using set
var count = 4;
set("name" + count, "Simone");    // Also sets name4 to "Simone"
trace (name4);
```

Note that, unlike its JavaScript cousin, *eval ( )* in ActionScript does not allow for the compiling and execution of arbitrary blocks of code in a string. Full support of *eval ( )* would require an ActionScript compiler in the Player, which would caus too great an increase in the Player size.

Usage

As of Flash 5, *eval ( )* is rarely needed for dynamic variable access. When managing multiple pieces of data, use arrays ar objects instead.

Example

The *eval ( )* function is often used to convert the `MovieClip ._droptarget` string property or the *Selection.getFo ( )* return value to an object reference. In the following example, suppose we have a series of cartoon face clips. When the drops a `food` clip onto one of the faces, we retrieve the path to the face in question using `_droptarget` , and then we *eval ( )* to retrieve an object reference to that face. Finally, we send the face to a frame showing the mouth full of food:

```
// Convert _droptarget string to a reference
theFace = eval(food._droptarget);
// Control appropriate face clip using converted reference
theFace.gotoAndStop("full");
```

See Also

"Executing Code in a String with eval," in Chapter 4

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Chapter 18. ActionScript Language Reference

_focusrect Global Property — Flash 4

the highlight state used for buttons and movie clips activated via the keyboard — ***read/write***

```
_focusrect
```

Description

When the mouse pointer hovers over a button in Flash, the content of the button's *Over* state is displayed. Buttons can also gain keyboard focus when the user presses the Tab key. When a button has keyboard focus, Flash places a yellow rectangle around that button, which is not always aesthetically desirable. You can turn off the yellow rectangle using the `_focusrect` global property, like so:

```
_focusrect = false;
```

When `_focusrect` is set to `false`, Flash displays the *Over* state of buttons with keyboard focus, or the frame labeled `_over` in movie clips with keyboard focus. When `_focusrect` is set to `true` (its default), Flash displays the yellow rectangle in addition to displaying the *Over* state (or `_over` frame).

Note that `_focusrect` can be set for individual button or movie clip instances via `Button._focusrect` and `MovieClip._focusrect`.

See Also

`Button._focusrect`, `MovieClip.focusEnabled`, `MovieClip._focusrect`, `MovieClip.tabEnabled`, *Selection.getFocus( )*, *Selection.setFocus( )*

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Chapter 18. ActionScript Language Reference

fscommand( ) Global Function — **Flash 3; enhanced in Flash 5 to support "trapallkeys"**

---

send a message to the Standalone Player or to the Player's host application (usually a web browser)

```
fscommand(command, parameters)
```

Arguments

command

A string passed to the host application, often the name of a JavaScript function.

parameters

A string passed to the host application, often an argument to the function named by `command` .

Description

With the *fscommand ( )* function, a Flash movie can communicate with the Standalone Player or with the Player's host application—the environment in which the Flash Player is running (usually a web browser, but possibly Macromedia Dire application that can host ActiveX controls). The *fscommand ( )* function typically is used in one of three ways:

- To send one of a limited set of built-in commands to the Standalone Player
- To send commands to a scripting language, such as JavaScript or VBScript, in a web browser
- To communicate with Lingo in a Macromedia Director movie

When used with the Standalone Player, *fscommand ( )* takes one of the built-in sets of command/argument pairs, as showı 18-6. Note that as of Flash 6, the `Stage.scaleMode` and `Stage.showMenu` properties are preferred over the `"allowscale"` and `"showmenu"` *fscommand ( )* calls.

Table 18-6. Command/argument pairs in Standalone Player

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Command	Argument	Description
"allowscale"	"true" or "false"	When "false ", prevents the contents of a movie from growing or shrin relation to the window size of the Player. "allowscale " is often used combination with "fullscreen " to create a Standalone Player that co entire screen while maintaining the movie's original size. See also Stage.scaleMode .
"exec"	"application_name "	Launches an external application (works in Standalone Player only). The p application is specified in the string *application_name* . In Flash M application *must* reside in a folder named *"fscommand"* directly beneath th (if a *.swf* file resides in */demo/* , the application to launch must reside in */demo/fscommand/*). For example, to launch *someApp.exe* , use *"fscommand("exec", "someApp.exe")"*, without the subfolder name */fscon* Attempts to use "exec" with an executable in another directory will fail In Flash 5, the application to launch can reside anywhere on the drive fron the movie was loaded. For details on forming paths for Flash 5 Projectors, http://www.macromedia.com/support/flash/ts/documents/fscommand_proj
"fullscreen "	"true " or "false "	When "true ", causes the Standalone Player's window to maximize (fill screen).
"quit "	not applicable	Closes the movie and exits the Standalone Player.
"showmenu "	"true " or "false "	When "false ", suppresses the display of the controls in the contextual the Player, leaving only minimal options. The contextual menu is accessed click on Windows and Ctrl-click on Macintosh. As of Flash 6, the Stage.showMenu is preferred.
"trapallkeys "	"true " or "false "	When "true ", causes all keystrokes—even keyboard shortcuts—to be s Flash movie. "trapallkeys " is used to disable the control keys in the Standalone Player (e.g., Ctrl-F or Command-F for Full Screen mode, Ctrl- Command-Q for Exit, Esc for Stop/exit Full Screen mode, etc.). Added to *fscommand* in Flash 5.

When used from within a *.swf* file playing in a browser, *fscommand ( )* invokes a special JavaScript function (in Netscape VBScript function (in Internet Explorer) on the HTML page that contains the movie. The name of this special function tak general form *movieID _DoFSCommand ( )* , where *movieID* is the name specified in the HTML <OBJECT> tag's attribute (in Internet Explorer) or the <EMBED> tag's NAME attribute (in Netscape). In the following Example, the <OBJ ID attribute is "testmovie" , so Internet Explorer executes the VBScript function *testmovie_DoFSCommand ( )* whe Player invokes an *fscommand ( )* .

When *movieID _DoFSCommand ( )* is invoked, the values of the *fscommand ( )* 's *command* and *parameters* a are passed as arguments to the *movieID _DoFSCommand ( )* function. If no *movieID _DoFSCommand ( )* functio the host HTML page, *fscommand ( )* fails silently. Although a detailed discussion of writing a *movieID _DoFSComm* function in JavaScript or VBScript is beyond the scope of this book, see the following Example for a sample.

Regardless, calls to *fscommand ( )* won't work unless your HTML page's <OBJECT> tag includes the ID attribute, or the <EMBED> tag includes the NAME attribute (the latter should not be confused with the unrelated NAME attributes used in <OBJECT> tags). These attributes are not included by default but can be added easily in the authoring tool under File Settings HTML Template Flash with FSCommand. See the Example for sample <OBJECT> and <EMBE Note that in order for *fscommand ( )* to work with Netscape, the SWLIVECONNECT attribute of the HTML <EMBED> t set to "true" (and don't forget the NAME attribute, which is also required) as shown in the Example.

Usage

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

It is *not* possible to communicate with a browser via *fscommand ( )* under the following system configurations:

- Internet Explorer on the Macintosh OS
- Any version of Netscape 6 prior to 6.2 (*fscommand( )* in Netscape 6.2 and later requires at least Flash Player 6.0.4(
- Any browser running on a 68K-series Macintosh
- Any browser running on Windows 3.1

Note that *fscommand ( )* does not always provide the best means of communicating with a Director movie from Flash. Th way to communicate with Macromedia Director is via the *getURL ( )* function using either the `event` : or `lingo` : prot details, see the *getURL ( )* function or the following Macromedia technotes:

http://www.macromedia.com/support/director/ts/documents/flash_xtra_sending_events.htm
http://www.macromedia.com/support/director/ts/documents/flash_xtra_lingo.htm
http://www.macromedia.com/support/director/ts/documents/flash_tips.htm

Example

To quit a Standalone Player (Projector), use:

```
fscommand("quit");
```

To run a Standalone Player fullscreen, use:

```
fscommand("fullscreen", "true");
```

To run a Standalone Player fullscreen while maintaining the original movie's size, use:

```
fscommand("fullscreen", "true");
fscommand("allowscale", "false");
```

For information on launching a movie in a fullscreen web browser window, see:

http://www.moock.org/webdesign/flash/launchwindow/fullscreen.

The following code shows an HTML page with the JavaScript and VBScript needed to respond to a simple *fscommand ( )* movie. Notice that the VBScript function simply calls the JavaScript function; this allows us to handle both Internet Explo Netscape with a single JavaScript function. Note the `ID` , `NAME` , and `SWLIVECONNECT` attributes in **`bold`** :

```
<HTML>
<HEAD>
<TITLE>fscommand demo</TITLE>

<SCRIPT LANGUAGE="JavaScript">
<!--
function testmovie_DoFSCommand(command, args) {
  alert("Here's the Flash message " + command + ", " + args);
}
//-->
</SCRIPT>
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

```
<SCRIPT LANGUAGE="VBScript">
<!--
Sub testmovie_FSCommand(ByVal command, ByVal args)
    call testmovie_DoFSCommand(command, args)
end sub
//-->
</SCRIPT>

</HEAD>

<BODY BGCOLOR="#FFFFFF">

<OBJECT
  ID="testmovie"
  CLASSID="clsid:D27CDB6E-AE6D-11cf-96B8-444553540000"
  WIDTH="100%"
  HEIGHT="100%"
  CODEBASE="http://download.macromedia.com/pub/shockwave/cabs/flash/swflash.
  <PARAM NAME="MOVIE" VALUE="flash-to-javascript.swf">

  <EMBED
    NAME="testmovie"
    SRC="flash-to-javascript.swf"
    WIDTH="100%"
    HEIGHT="100%"
    SWLIVECONNECT="true"
    PLUGINSPAGE="http://www.macromedia.com/go/flashplayer/"
  </EMBED>
</OBJECT>

</BODY>
</HTML>
```

To invoke the preceding *testmovie_DoFSCommand ( )* JavaScript function from the *flash-to-javascript.swf* movie, we use

```
fscommand("hello", "world");
```

See Also

For more information on *fscommand ( )* and controlling Flash with JavaScript, see:

http://www.moock.org/webdesign/flash/fscommand
http://www.macromedia.com/support/flash/publishexport/scriptingwithflash

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Chapter 18. ActionScript Language Reference

Function Class Flash 6

object-oriented representation of ActionScript functions

Properties

prototype

An object specifying methods and properties for a class.

Methods

apply( )

Invoke a function as an object method, passing parameters in an array.

call*( )*

Invoke a function as an object method.

toString*( )*

Returns a string representation of the function object.

Description

In ActionScript, every function, both user-defined and built-in, is an instance of the *Function* class. As an object, every fu can have properties and methods, and every function can be stored, passed, and compared as ActionScript data. While functions-as-objects is a foreign (and perhaps strange) concept to some programmers, it is a core component of the ECMA implementation of object-oriented programming. In ActionScript, every function has the potential to be used as a class constructor. Hence, every function defines a `prototype` property specifying the methods and properties that will be inherited by the instances of the class. Furthermore, every function has the potential to be used as a method. Hence, every function can be stored as an object property (defining it as a method for that object). These and other uses of functions in ActionScript's inheritance system are discussed in Chapter 12.

Although functions were objects in Flash 5, it was not until Flash 6 that the *Function* class was formally exposed in ActionScript (at which time the *apply( )* and *call( )* methods were added, providing specialized tools for invoking function methods of arbitrary objects).

Some commands originally implemented in Flash 4 (e.g., *getURL( )* and *trace( )*) work like functions but are actually compiled to their own bytecodes, are not members of the *Function* class, and do not have `prototype` properties.

For general information on creating and using functions in ActionScript, see Chapter 9.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Usage

Note that new *Function* objects cannot be created with the *Function* constructor in ActionScript. All functions must be def with either the *function* statement or a function literal. For example, the following code is legal in JavaScript, but not in ActionScript:

```
var divide = new Function ("numerator", "divisor", "return numerator/divisor
```

See Also

The *Arguments* object; Chapter 9, Chapter 12

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Function.apply( ) Method — Flash 6

invoke a function as an object method, passing parameters in an array

```
theFunction.apply(thisObj, parametersArray)
```

Arguments

thisObj

The object on which `theFunction` is called as a method.

parametersArray

An array of values passed as arguments to `theFunction`.

Returns

The return value of `theFunction`.

Description

The *apply( )* method invokes `theFunction` as a method of `thisObj` and passes the values of `parametersArray` to `theFunction` as arguments. Within `theFunction`, the value of the `this` keyword is a reference to `thisObj`. The return value of *apply( )* is simply the return value of `theFunction`. To invoke `theFunction` without specifying an object reference, use `null` as the value of `thisObj`. To pass a comma-delimited list of arguments to `theFunction`, instead of passing the values in a `parametersArray`, use *Function.call( )*.

The *apply( )* method is normally used to alter the value of the `this` keyword when running a function, or to supply an array of arguments to a function in place of a comma-delimited list.

The following code shows the basic syntax of *apply( )*:

```
// Create a function
function square (x) {
  return x*x;
}

// Invoke the function as a method of _root, passing an array literal
// containing the single argument value 10.
tenSquared = square.apply(_root, [10]);
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

This example has little practical purpose—it would be easier to invoke the function directly, as in: `_root.square(10);`. The *apply( )* method becomes more useful when we need to dynamically specify the object on which to invoke a method, for example, when we want to use one method with many different objects that do not share a common class or superclass. Suppose we have *getWidth( )* and *moveClipTo( )* methods that we want to use with both a rectangle and a circle object. Here's the code we can use to create a generic method, *Rectangle.invokeAsMethod( )*, that invokes *getWidth( )*, *moveClipTo( )*, or any other function as a method of a rectangle object:

```
// A Rect class
function Rect (width, height, x, y) {
  this.width = width;
  this.height = height;
  this.x = x;
  this.y = y;
}

// Method to invoke any passed func as a method of a Rect object.
// Note that func is a function reference, not a string.
Rect.prototype.invokeAsMethod = function (func) {
  // Invoke func as a method of this Rect, passing along any
  // arguments that were supplied.
  func.apply(this, arguments.slice(1));
};

// A simple generic getWidth function that takes no arguments
function getWidth () {
  trace("width: " + this.width);
  return this.width;
}

// A slightly more complex example, moveClipTo, that takes arguments
function moveClipTo (newX, newY) {
  this.x = newX;
  this.y = newY;
}

// Make a new Rect object
box = new Rect(15, 20);
// Invoke getWidth on box
box.invokeAsMethod(getWidth);  // Displays: width: 15
// Invoke moveClipTo on box, passing the newX and newY arguments
box.invokeAsMethod(moveClipTo, 100, 200);
// Check if it worked...
trace("box x: " + box.x);  // Displays: box x: 100
trace("box y: " + box.y);  // Displays: box y: 200
```

This practice is not necessarily recommended when the objects can logically be assigned a common class or superclass. For example, rather than use *apply( )* in our rectangle example, it's more appropriate to create a *Shape* superclass that defines *moveClipTo( )* and *getWidth( )* methods inherited by both `Rect` and `Circle`. For a more legitimate application of the *apply( )* method, see the *CountDown* class shown in the following Example.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Example

The following class, *CountDown*, invokes a specified object's method after a delay. It uses *apply( )* to ensure that the method executes on the correct object and to pass the method arguments as a convenient array.

```
/*
 * CountDown Class
 *   Desc: Calls a method after delay milliseconds.
 * Params: obj  An object whose method will be called.
 *         meth  The string method name to call.
 *        delay  The time (milliseconds) to wait before calling.
 * arg4...argn  The arguments to pass to meth.
 */
function CountDown (obj, meth, delay) {
  // Remember how long to wait before calling meth.
  this.delay = delay;
  // Store a reference to the method.
  this.callback = obj[meth];
  // Store a reference to the object.
  this.obj = obj;
  // Store extra arguments. We'll pass them to meth later.
  this.args = arguments.slice(3);
  // Set an empty intervalID. Used with setInterval().
  this.intervalID = -1;
  // Start the countdown
  this.start();
}

/*
 * Method: start()
 */
CountDown.prototype.start = function () {
  // Invoke the method-caller after this.delay milliseconds.
  this.intervalID = setInterval(this, "invokeCallback", this.delay);
};

/*
 * Method: stop()
 */
CountDown.prototype.stop = function () {
  // Stop the interval so that it doesn't call the method again.
  clearInterval(this.intervalID);
  this.intervalID = -1;
};

/*
 * Method: invokeCallback()
 */
CountDown.prototype.invokeCallback = function () {
  // Time's up! Stop the countdown.
  this.stop();
  // Invoke the callback as a method of this.obj, and pass
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

```
    // along this.args to use as method arguments.
    this.callback.apply(this.obj, this.args);
};

// SAMPLE USE
// Create the countdown. Will invoke
// _root.output("hello", "world") after 1000 milliseconds.
count = new CountDown(_root, "output", 1000, "hello", "world");

// The method to invoke
function output (arg1, arg2) {
    trace("output: " + arg1 + ", " + arg2);
}
```

See Also

Function.call( ), the *Arguments* object; Chapter 12

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Function.call( ) Method Flash 6

invoke a function as an object method

```
theFunction.call(thisObj, param1, param2,...paramn)
```

Arguments

thisObj

The object on which `theFunction` is called as a method.

param1, ...paramn

A list of values passed as arguments to `theFunction`.

Returns

The return value of `theFunction`.

Description

The *call( )* method invokes `theFunction` as a method of `thisObj`, and passes the values of `param1`, `param2`, `...paramn` to `theFunction` as arguments. It is functionally identical to the *apply( )* method, except that it specifies arguments to `theFunction` as a comma-delimited list, rather than as an array. For complete details, see *Function.apply( )*.

Example

```
function moveClipTo (newX, newY) {
  this._x = newX;
  this._y = newY;
}
moveClipTo.call(ball_mc, 100, 200);  // Moves ball_mc to 100, 200
```

See Also

Function.apply( ), the *Arguments* object; Chapter 12

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Chapter 18. ActionScript Language Reference

Function.prototype Property — Flash 5

an object specifying methods and properties for a class — ***read/write***

theFunction.prototype

Description

The `prototype` property is the centerpiece of ActionScript's class and object inheritance system. By default, it is a generic object waiting to be assigned properties and methods that will become available to objects constructed by *`theFunction`*. Like any generic object, *`theFunction`*`.prototype` starts off with the built-in properties `_ _proto_ _` and `constructor`. ActionScript automatically ensures that *`theFunction`*`.prototype.constructor` refers to *`theFunction`* (in all other situations, an object's constructor property initially refers to its own constructor function, which in the case of a generic object would be *Object*, not *`theFunction`*).

The `prototype` object can also be replaced by a new object, as is commonly done when specifying a class's superclass.

The *Function* class itself is also a function, so it too defines a `prototype` property that can be used to add methods and properties to all functions. For example, the following code adds a *getName( )* and *setName( )* method to all functions, and overrides the default *Function.toString( )* method:

```
Function.prototype.getName = function () {
  return this.funcName;
};
Function.prototype.setName = function (name) {
  this.funcName = name;
};
Function.prototype.toString = Function.prototype.getName;

// Usage:
function squareNum (x) {
  return x * x;
}

squareNum.setName("squareNum");
trace(squareNum);  // Displays: squareNum
                   // (instead of [type Function])
```

For a full discussion of `prototype`'s role in object-oriented ActionScript programming, see Chapter 12. Within the advanced developer community, the `prototype` property is the focus of a lively debate on

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

superclass assignment methodology. See Section 12.7.3.

See Also

Object.constructor, *Object*._ _proto_ _; Chapter 12

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Function.toString( ) Method Flash 5

returns a string representation of the function object

```
theFunction.toString()
```

Returns

A string representing the function object.

Description

The *toString( )* method returns a string describing `theFunction`. By default, all functions return the string "[type Function]". Checking the return value of *toString( )* can be useful during debugging, as it distinguishes function objects from other objects. The *instanceof* and *typeof* operators can also be used to determine the datatype of a *Function* object. For example:

```
function moveClipTo (newX, newY) {
  this._x = newX;
  this._y = newY;
}
trace(moveClipTo.toString());           // Displays: [type Function]
trace(typeof moveClipTo);               // Displays: function
trace(moveClipTo instanceof Function);  // Displays: true
```

To provide a more descriptive return value for *toString( )*, you can override the *toString( )* method for the function:

```
moveClipTo.toString = function () {
  return "moveClipTo function";
};

trace(moveClipTo.toString());  // Displays: moveClipTo function
```

See Also

instanceof, *Object.toString( )*, *typeof*

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Chapter 18. ActionScript Language Reference

getProperty( ) Global Function — Flash 5

retrieve the value of a movie clip propertyFlash 4; deprecated in

`getProperty(`*`movieClip, property`*`)`

Arguments

movieClip

An expression that yields a string indicating the path to a movie clip. As of Flash 5, this can also be a movie clip reference, because movie clip references are converted to paths when used in a string context.

property

The name of the built-in property to retrieve. Must be an identifier, not a string (e.g., `_x`, not `"_x"`).

Returns

The value of *`movieClip`*'s *`property`*.

Description

The *getProperty( )* function retrieves the value of one of a movie clip's built-in properties. Though *getProperty( )* was the only way to access object properties in Flash 4, the dot and [ ] operators are the preferred property-access tools as of Flash 5.

Example

Each of the following *getProperty( )* invocations retrieve the values of the `_x` property of a movie clip, named `ball`, on the main movie timeline:

```
getProperty(ball, _x);              // Relative movie clip reference
getProperty(_root.ball, _x);        // Absolute movie clip reference
getProperty("ball", _x);            // Relative path in string
getProperty("_root.ball", _x);      // Dot path in string
getProperty("/ball", _x);           // Slash path in string
```

The following code shows similar property access using the preferred dot and [ ] operators:

```
ball._x;
_root.ball._x;
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

```
ball["_x"];
_root["ball"]["_x"];
_root["ball"]["_x"];
```

See Also

setProperty( ); Section 13.1

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Chapter 18. ActionScript Language Reference

getTimer( ) Global Function — Flash 4

determine how long the Player has been running, in milliseconds

```
getTimer()
```

Returns

The number of milliseconds that have elapsed since the Player started running.

Description

The *getTimer( )* function indicates how long the Player has been running, in milliseconds. The *getTimer( )* function return a single value, per Player instance, for all movies in the Player. Loading a new *.swf* does not reset the timer. Prior to Flash 6, developers used multiple calls to *getTimer( )* to govern the timed execution of a block of code, or to add time-based features to a movie, such as a countdown in a video game. As of Flash 6, timed code execution should be implemented using *setInterval( )* .

The following code shows a countdown timer function, *doCountdown( )* , that displays the seconds remaining on screen. For a more sophisticated *CountDown* class that uses *setInterval( )* , see *Function.apply( )* .

```
function doCountdown (countAmount) {
  // Record the starting time, in milliseconds
  var startTime = getTimer();
  // Initialize a variable to keep track of how much time has passed
  var elapsed = 0;
  // Create a text field to display the countdown time in seconds
  this.createTextField("counterOutput_txt", 1, 0, 0, 200, 20);

// With each frame that passes...
this.onEnterFrame = function () {
  // ...check how much time has passed
  elapsed = getTimer() - startTime;
  // If the elapsed time is less than the length of our countdown...
  if (elapsed < countAmount * 1000) {
    // ...set the text field to show the number of seconds left
    this.counterOutput_txt.text = countAmount - Math.floor(elapsed / 1000)
  } else {
    // ...otherwise, our countdown is done, so tell the user
    this.counterOutput_txt.text = "Time's Up!";
    // Stop checking the time
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

```
      delete this.onEnterFrame;
    }
  }
}

// Usage:
doCountDown(60);
```

Example

To determine the number of full seconds (rather than milliseconds) that have elapsed during a movie, divide the return value of *getTimer( )* by 1000, and trim off the decimal portion with either *Math.floor( )* , *Math.round( )* , or *Math.ceil( )* . For example:

```
numSeconds = Math.floor(getTimer()/1000);
```

The following code loops between two frames until a movie has been playing for more than 10 seconds, after which it breaks out of the timeline loop:

```
now = getTimer();
if (now > 10000) {
  play();
} else {
  gotoAndPlay(_currentframe - 1);
}
```

The following code does the same thing using *setInterval( )* :

```
stop();
intID = setInterval(function () { play(); clearInterval(intID); }, 1000);
```

See Also

clearInterval( ) , *Date( )* , the *Date* class, *setInterval( )*

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Chapter 18. ActionScript Language Reference

getURL( ) Global Function — Flash 4

load a document into a browser, execute server-side scripts, or trigger Macromedia Director eventsFlash 2; method parameter added in

```
getURL (url)
getURL (url, window)
getURL (url, window)
getURL (url, window, method)
```

Arguments

url

A string specifying the absolute or relative location of the document to load or external script to run, or a commanc (depending on the protocol, as shown in Table 18-7).

window

An optional string specifying the name of the browser window or frame into which to load the document. Can be a custom name or one of the four presets: `"_blank"` , `"_parent"` , `"_self"` , or `"_top"` .

method

An optional string specifying the method by which to send the current timeline's variables to an external script—ei `"GET"` or `"POST"` . This parameter must be a literal string, not a variable or other expression. On Windows, the Standalone Player always uses the `"GET"` method, regardless of the *`method`* specified. See Appendix F for det on support for GET and POST.

Description

The *getURL( )* function is used to:

- Load a document (usually a web page) into a web browser frame or window
- Execute a server-side script and receive the results in a browser frame or window
- Execute JavaScript code in a web browser
- Trigger events from Flash assets imported as sprites into Macromedia Director

To load a document into the current window or frame, simply specify the URL of the document as the first parameter, *`ur`* without supplying a *`window`* or *`method`* argument. Naturally, Flash supports absolute URLs (those that contain a pro

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

such as "*http:* " plus a server name or hardware device) and relative URLs (those that are relative to the current location):

```
getURL("http://www.moock.org/");                    // Absolute URL to web pag
getURL("file:///C:/WINDOWS/Desktop/index.html");    // Absolute URL to local f
getURL("/whatever/index.html");                     // Relative URL, http prot
                                                    // is assumed
```

To load a document into a named window or frame, supply the window or frame name as the *window* argument. For example:

```
getURL("http://www.moock.org/", "contentFrame");  // Load into named frame
getURL("http://www.moock.org/", "remoteWin");     // Load into named window
```

To replace the frameset that contains the current movie, use "_parent" as the value of the *window* argument. For example:

```
getURL("http://www.moock.org/", "_parent");
```

To replace *all* framesets in a web page with a loaded document, use "_top" as the value of the *window* argument. Fo example:

```
getURL("http://www.moock.org/", "_top");
```

To open a loaded document in a new, anonymous browser window, use "_blank" as the value of the *window* argum For example:

```
getURL("http://www.moock.org/", "_blank");
```

Note that launching a new window with "_blank" does not give us any control over the appearance of the new windov (e.g., its size, toolbar configuration, location, etc.). To launch customized windows with *getURL( )* , we must invoke a JavaScript function on the movie's host page. Techniques for launching windows with JavaScript are described at:

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

Prior to Flash 6, the *getURL( )* function was used to send variables to a remote server application or script, and receive re in a web browser. As of Flash 6, the *LoadVars.send( )* method is better suited for that task. To send the current movie clip timeline variables to an external script using getURL(), specify the name of the script as the *url* argument, and use eithe "GET" or "POST" as the *method* argument. For example:

```
getURL("http://www.someserver.com/cgi-bin/search.pl", "resultsFrame", "GET")
```

When invoked as a movie clip method, *getURL( )* sends the timeline variables of that clip to the remote script, as in:

```
// Sends myClip's variables to search.pl
myClip.getURL("http://www.server.com/cgi-bin/search.pl", "resultsFrame", "GE
```

The results of the script execution will appear in the window or frame specified by *getURL( )* 's *window* argument (whi required when variables are submitted).

To load the results of a script execution into the current frame or window, use "_self" as the *window* argument valu in:

```
getURL("http://www.someserver.com/cgi-bin/search.pl", "_self", "GET");
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

When the *method* argument is "GET", the current movie clip's timeline variables are sent as a query string attached t script URL in an HTTP GET request. Query strings are composed of variable name/value pairs, separated by ampersands For example:

```
http://www.someserver.com/cgi-bin/search.pl?term=javascript&scope=entiresite
```

When the *method* argument is "POST", the current movie clip's timeline variables are sent to the script as a separate of data after the HTTP POST-request header (exactly like a regular HTML form that uses the POST method). Note that o Windows, "POST" is not available in the Standalone Player.

Because many web servers restrict the length of URLs to between 255 and 1024 characters, use "POST" instead of "GE transfer larger amounts of data.

Note that any information returned by a script invoked by *getURL( )* is displayed as regular web content in the browser, n Flash. To accept the results of a script execution into Flash, use *loadVariables( )* or, preferably, the *LoadVars* class. To lo *.swf* file, see the *loadMovie( )* function.

The *getURL( )* function can also be used with protocols other than "*http:* ", as shown in Table 18-7.

Table 18-7. Supported protocols for getURL( )

Protocol	Format	Purpose
event	"event: *eventName params* "	Send an event to Director if the Flash asset is a Director sprite.
file	"file:///driveSpec/folder/filename"	Access a local file.
ftp	"ftp://server.domain.com/folder/filename"	Access a remote file via FTP (file transfer protocol).
http	"http://server.domain.com/folder/filename"	Access remote file via HTTP (hypertext transfer protocol).
javascript	"javascript: *command* "	Perform JavaScript command in browser.
lingo	"lingo: *command* "	Perform Lingo command if Flash asset is a Director sprite.
print	"print:", "*targetClip* "	Print the target clip. In Flash 5 or later, use *print( )* instead.
vbscript	"vbscript: *command* "	Perform VBScript command in browser.
mailto	"mailto:user@somedomain.com"	Send email via the default mail program on the user's system.
telnet	"telnet://domain.com:8888/"	Telnet to domain.com on port 8888.

As shown in Table 18-7, if a Flash asset is imported into a Macromedia Director file, *getURL( )* can be used to trigger Li events or execute a Lingo command. (Lingo is Director's scripting language, akin to ActionScript). For example, you can frame event of this form:

```
getURL ("event: eventName params");
```

which will cause the Lingo event handler named *on eventName* to be called in Director. Here is a *getURL( )* stateme that generates an event named "myEvent" and passes it the string "A". Note that the " character is escaped using the seque \" :

```
getURL ("event: myEvent \"A\"");  // Send an event to Director
```

Here is the Lingo *sprite* event handler that should be attached to the Flash sprite asset in Director in order to receive the ev Note that a Director sprite is roughly equivalent to a Flash movie clip instance; the Lingo keyword *put* is akin to ActionSc *trace( )* command, and && is Lingo's string concatenation operator:

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

```
on myEvent msg
  put "The message received from Flash was " && msg
end
```

You can also trigger Lingo to be executed from a Flash sprite within Director using the `"lingo:"` keyword, such as:

```
getURL ("lingo: beep");  // Tell Director to play a beep sound
```

Using Flash assets in Macromedia Director requires an appropriate version of the so-called *Flash Asset Xtra* , a free plugin for Director, provided by Macromedia. Director 8.5 includes the Flash Asset Xtra for Flash 5 *.swf* files, but it cannot import Flash 6 *.swf* files. An updated Flash 6 (Flash MX) Asset Xtra ships with Director MX (i.e., Director 9) and is also available from Macromedia's site. The Flash MX Asset Xtra also works with Shockwave 8.5.1, but requires Mac OS 8.6 or later, or Windows 98 or later.

Finally, *getURL ( )* can also be used to execute JavaScript code. Here we invoke a simple JavaScript *alert* using *getURL (*

```
getURL ("javascript: alert('hello world');");
```

Note that execution of JavaScript code from a URL does not work in Internet Explorer 4.5 for Macintosh.

Example

Here's the code for a standard button that links to a web page:

```
link_btn.onRelease = function () {
  getURL("http://www.macromedia.com/");
}
```

Bugs

Internet Explorer 4.5 (IE 4.5) and older versions on Macintosh do not support the `"POST"` method of a *getURL( )* call. service those browsers, use `"GET"` instead of `"POST"` (subject to the length limitations cited earlier).

In most browsers, relative URLs passed to *getURL( )* are resolved relative to the HTML file that contains the *.swf* file. In 4.5 and older versions on Macintosh, relative links are resolved relative to the location of the *.swf* file, not the HTML fil which causes problems when the two are in different directories. To avoid the problem, either place the *.swf* and *.html* file the same directory or use absolute URLs when invoking *getURL( )* , such as:

```
getURL ("http://www.someserver.com/")
```

See Also

loadMovie( ) , *loadVariables( )* , the *LoadVars* class, *MovieClip.getURL( )* , *MovieClip.loadVariables( )* ; "< A> (Anchor Hypertext Link)" in Appendix E

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

getVersion( ) Global Function Flash 5

examine the platform and version of the Flash Player

```
getVersion( )
```

Returns

A string containing version and platform information for the Flash Player hosting the current movie.

Description

The *getVersion( )* function tells us the platform and Flash Player version being used to view a movie. It can be used to execute different code conditionally for specific versions of the Flash Player or on certain operating systems. As of Flash 6, the `capabilities.version` property is preferred over *getVersion( )*. However, for movies running in Flash Player 5, *getVersion( )* must still be used. In Flash 4, the global `$version` property must be used.

See `capabilities.version` for sample code and a complete description of the possible return values of *getVersion( )*.

See Also

`capabilities.version`, `$version`; Appendix C

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Chapter 18. ActionScript Language Reference

_global Object Flash 6

container for global variables

`_global.variableName`

Description

The *_global* object enables the creation of global variables (i.e., variables that are available from the timeline of any movie clip or level). Unlike in JavaScript, in ActionScript there is no "global execution context" where variable definitions automatically create global variables. All nonlocal variables created in ActionScript are scoped to the movie clip in which they are created. To make a variable directly accessible throughout all timelines in a movie, we must deliberately add a property to `_global` . For example:

```
// Add a property, age, to _global. Creates a global variable, age.
_global.age = 31;
// Access age from any timeline:
trace(age);  // Displays: 31
```

Using global variables, we can make a function, property, or object available globally, much like the built-in classes and objects (*Math* , *TextField* , *Mouse* , etc.). For example, if we want to make the *CountDown* class a global class, we can use the following code (see *Function.apply( )* for the full listing for *CountDown*):

```
_global.CountDown = function (obj, meth, delay) {
  // ... Class code here
};
```

Then, from any timeline, we can create instances of *CountDown* directly, as in:

```
theTimer = new CountDown(game, "timesUp", 10000);
```

Global variables do not become properties of all objects. For example, you cannot access the global variable `age` as a property of a *Date* object:

```
_global.age = 31;  // Make the global variable
now = new Date();  // Make the Date object
trace(now.age);    // age is not a property of now. Displays: undefined
```

Similarly, global variables cannot be accessed on movie clips via property-style dot notation. For example, a global variable named `password` cannot be accessed as `form_mc` .`password` (where `form_mc` is a movie clip). Similarly, from `form_mc` 's timeline, `password` cannot be accessed as `this.password` . The global variable `password` can be accessed only as `_global.password` or directly as `password` (without any qualifying object reference) from the current timeline.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Assigning a timeline variable with the same name as a global variable overrides (but does not overwrite!) the value of the global variable. For example, if a movie clip named `player1_mc` assigns the variable `age` a value of 10, then `age` will contain 10 inside `player1_mc` :

```
// CODE ON player1_mc TIMELINE
_global.age = 31;  // Create a global variable named age
var age = 10;      // Create a timeline variable also named age
trace(age);        // Displays: 10
```

But from any other timeline, the global variable `age` will still contain 31. In fact, if we remove the timeline variable `age` from `player1_mc` , references to `age` will then access the global variable:

```
delete age;
trace(age);        // Displays: 31
```

To assign or reassign the value of a global variable, we must use a fully qualified reference to the `_global` property, as in:

```
_global.age = 21;  // This change in age is visible from every timeline that
                   // does not specify its own value for age.
```

Local function variables similarly override globals:

```
_global.age = 21;  // Set global variable
function showAge () {
  var age = 5;     // Set local variable
  trace(age);
}
showAge();         // Displays: 5
                   // Local variable overrides global variable
trace(age);        // Displays: 21
                   // Global variable still exists outside the function
```

For more information on how variables override each other, see Section 9.7.1. To remove a global variable, use the *delete* operator, as follows:

```
delete _global.age;  // Deletes age. (The reference to _global is mandatory.
```

When using global variables, be careful to specify uniquely identifiable names, particularly in movies that will be used by other developers or loaded into their movies. One option is to preface variables with your domain name, as in:

```
_global.moockorgGravity = 9.8;
```

Or, for easier reading, create an object that contains all globals, as in:

```
_global.moockorg = new Object();
_global.moockorg.gravity = 9.8;
```

Java developers may even prefer:

```
// Create namespace.
if (!_global.org) {
  _global.org = new Object();
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

```
}
if (!_global.org.moock) {
  _global.org.moock = new Object();
}

// Assign variable in namespace.
_global.org.moock.gravity = 9.8;
```

To make globals easy to identify, consider using a naming convention such as the suffix `_gl` , as in:

```
_global.moockorgGravity_gl = 9.8;
```

Bugs

In Flash 6, a so called "getter/setter" property cannot be added directly to the `_global` object. See *Object.addProperty( )* for details.

See Also

Chapter 2; "The Scope Chain," in Chapter 9

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Chapter 18. ActionScript Language Reference

gotoAndPlay( ) Global Function — Flash 2

move the playhead to a given frame, and play the current clip or movie

```
gotoAndPlay(frameNumber)
gotoAndPlay(frameLabel)
gotoAndPlay(scene, frameNumber)
gotoAndPlay(scene, frameLabel)
```

Arguments

frameNumber

A positive integer indicating the number of the frame to which the playhead of the current timeline should jump before playing. If `frameNumber` is less than 1 or greater than the number of frames in the timeline, the playhead is sent to either the first or last frame, respectively.

frameLabel

A string indicating the label of the frame to which the playhead of the current timeline should jump before playing. If `frameLabel` is not found, the playhead is sent to the first frame of the current scene.

scene

An optional string indicating the name of the scene that contains the specified `frameNumber` or `frameLabel` . If not supplied, the current scene is assumed.

Description

When invoked without a `scene` argument, *gotoAndPlay( )* sends the playhead of the current timeline to the frame specified by either `frameNumber` or `frameLabel` and then plays the timeline from that point. The "current timeline" is the timeline of the movie clip or main movie from which the *gotoAndPlay( )* function is invoked.

When jumping to another frame, any intervening frames are skipped and their scripts are not executed.

If two arguments are specified in the *gotoAndPlay ( )* function call, the first argument is assumed to be the `scene` . If only one argument is specified, it is treated as a `frameNumber` or `frameLabel` , and the current scene is assumed.

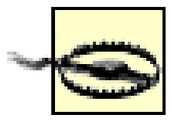
When invoked with a `scene` argument, *gotoAndPlay ( )* moves the playhead to the frame number or label in the

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

specified `scene` and then plays that scene. If a `scene` argument is used, the *gotoAndPlay ( )* function must be invoked from the `_root` timeline; otherwise, the operation fails silently and the playhead is not sent to the destination frame. Note that scenes are flattened into a single timeline during movie playback. That is, if scene 1's timeline contains 20 frames, and scene 2's timeline contains 10 frames, then we can send the playhead to frame 5 of scene 2 by placing the following code in scene 1:

```
gotoAndPlay(25);
```



I recommend against using scenes when working with ActionScript-intensive movies. Unlike movie clips, scenes are not represented by objects and cannot be manipulated directly by most built-in functions. It's normally better to use labels and movie clips as pseudo-scenes in your timeline, instead of using Flash's scene feature.

The global *gotoAndPlay( )* function affects the current timeline only. The frames or state of other movie clips within the current timeline are not affected. To cause other movie clips to play, you must issue a separate *play( )* or *gotoAndPlay( )* command for each movie clip, including nested ones. To apply the *gotoAndPlay( )* function to a clip other than the current movie clip, use the movie clip method form, *MovieClip.gotoAndPlay( )* .

Bugs

In Flash Player 5 and Flash Player 6, the *gotoAndPlay( )* function does not work when used in an *onClipEvent( )* handler with a string literal for `frameLabel` .

In Flash Player 4 and later, when a nonliteral `frameLabel` argument is specified in any scene other than the first, the playhead jumps to the specified label *plus* the total number of frames in all scenes before the current scene.

To work around both bugs, use the movie clip alternative, *MovieClip.gotoAndPlay( )* , and specify `this` to indicate the current clip, as in `this` *.gotoAndPlay("myLabel")* , rather than *gotoAndPlay("myLabel")* .

Example

```
// Go to frame 5 of the current timeline, and play it
gotoAndPlay(5);
// Go to frame 10 of the exitSequence scene, and play it
gotoAndPlay("exitSequence", 10);
// Go to frame "goodbye" of the exitSequence scene, and play it
gotoAndPlay("exitSequence", "goodbye");
// Caution! This plays the frame labeled "exitSequence" in the current scene
gotoAndPlay("exitSequence");
// This plays frame 1 of the exitSequence scene
gotoAndPlay("exitSequence", 1);
```

Reader Exercise: Write a function that recursively plays all nested movie clips within the clip from which *gotoAndPlay( )* is called. (Hint: Use a for-in loop to retrieve a list of child clips nested within each clip.)

See Also

gotoAndStop( ) , *MovieClip.gotoAndPlay( )* , *play( )* , *stop( )*

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Chapter 18. ActionScript Language Reference

gotoAndStop( ) Global Function — Flash 2

move the playhead to a given frame, and stop the current clip or movie

```
gotoAndStop(frameNumber)
gotoAndStop(frameLabel)
gotoAndStop(scene, frameNumber)
gotoAndStop(scene, frameLabel)
```

Arguments

frameNumber

A positive integer indicating the number of the frame to which the playhead of the current timeline should jump. If `frameNumber` is less than 1 or greater than the number of frames in the timeline, the playhead is sent to either the first or last frame, respectively.

frameLabel

A string indicating the label of the frame to which the playhead of the current timeline should jump. If `frameLal` is not found, the playhead is sent to the first frame of the current scene.

scene

An optional string indicating the name of the scene that contains the specified `frameNumber` or `frameLabe` If not supplied, the current scene is assumed.

Description

When invoked without a `scene` argument, *gotoAndStop( )* sends the playhead of the current timeline to the frame speci by either the `frameNumber` or `frameLabel` argument. The "current timeline" is the movie or movie clip from wh the *gotoAndStop( )* function is invoked. The playhead will stop at the target frame; it will not advance automatically after arriving at the target frame.

If two arguments are specified in the *gotoAndStop( )* function call, the first argument is assumed to be the `scene` . If on one argument is specified, it is treated as a `frameNumber` or `frameLabel` , and the current scene is assumed.

When invoked with a `scene` argument, *gotoAndStop( )* moves the playhead to the frame number or label in the specifie scene and then halts playback. If a `scene` argument is used, the *gotoAndStop( )* function *must* be invoked from the `_r` timeline; otherwise, the operation fails silently and the playhead is not sent to the destination frame.

The global *gotoAndStop( )* function affects only the current timeline. The frames or state of other movie clips within the current timeline are not affected. To move the playhead of other movie clips, you must issue a separate *gotoAndStop( )*

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

command for each movie clip. To apply the *gotoAndStop( )* function to a clip besides the current movie clip, use the movi clip method form, *MovieClip .gotoAndStop( )* .

Bugs

In Flash Player 5 and Flash Player 6, *gotoAndStop( )* does not work when used in an *onClipEvent( )* handler with a string literal for *frameLabel* .

In Flash Player 4 and later, when a nonliteral *frameLabel* argument is specified in any scene other than the first, the playhead jumps to the specified label *plus* the total number of frames in all scenes before the current scene.

To work around both bugs, use the movie clip variation of *gotoAndStop( )* and specify `this` to indicate the current clip, in *this .gotoAndStop("myLabel"*), rather than *gotoAndStop("myLabel")* .

Example

```
// Go to frame 5 of the current timeline, and stop there
gotoAndStop(5);
// Go to frame 20 of the introSequence scene, and stop there
gotoAndStop("introSequence", 20);
// Go to frame "hello" of the introSequence scene, and stop there
gotoAndStop("introSequence", "hello")
// Caution! This goes to the frame labeled "introSequence" in the current sc
gotoAndStop("introSequence")
// This goes to frame 1 of the introSequence scene
gotoAndStop("introSequence", 1)
```

Reader Exercise: Write a custom function, *gotoDeltaAndStop( )* , that allows the caller to specify a relative frame number including a negative one. For example, *gotoDeltaAndStop(-5)* would jump backward by 5 frames. (Hint: You'll need the *MovieClip* ._currentframe property.) Remember to add any special code to handle the end conditions (i.e., whe relative frame number would cause it to jump beyond the beginning or end of the current timeline).

See Also

gotoAndPlay( ) , *MovieClip.gotoAndStop ( )* , *play( )* , *stop( )*

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Chapter 18. ActionScript Language Reference

_highquality Global Property

the rendering quality of the Player ***read/write***

`_highquality`

Description

The `_highquality` global property stores an integer between 0 and 2 that dictates the rendering quality of the Flash Player as follows:

0

Low quality. Neither bitmaps nor vectors are antialiased (smoothed).

1

High quality. Vectors are antialiased. Bitmaps are antialiased when no animation is occurring.

2

Best quality. Both bitmaps and vectors are always antialiased.

As of Flash 5, `_highquality` has been superceded by `_quality`, which can be used to set a movie's quality to "Medium", "AutoLow", and "AutoHigh", as well as "Low", "High", and "Best".

See Also

`_quality`, *toggleHighQuality( )*

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Chapter 18. ActionScript Language Reference

#include Directive — Flash 5

import the text of an external ActionScript file

```
#include path
```

Arguments

path

A string indicating the name and location of the script file to import, which can be specified relative to the *.fla* file as an absolute path (see samples under Example). Note that forward slashes, not backslashes, should be used in the path. Script files should be named with the *.as* file extension.

Description

The `#include` directive imports script text from an external text file (preferably one with the *.as* extension) into the current script, placing it directly where the `#include` command occurs in the script. If the file can't be found at the specified `path`, the directive will cause an error and no external text will be included.

The `#include` operation is performed at compile time, meaning that the text included in a movie is the text that existed the time the movie was tested, exported, or published from the authoring tool. If the external file changes after the movie i exported, the changes will not be reflected in the movie. In order for the changes to be incorporated into the movie, the movie must be reexported.

> Including an external file with the `#include` directive does not alter the *.fla* file permanently; nor does it allow you to view the imported file's contents easily. Likewise, if the external file is missing, or if it has changed, the next time the *.fla* file is recompiled into a *.swf*, your code may break unexpectedly. To permanently import an external *.as* file's code and break the connection to the external *.as* file itself, you must copy and paste the text into the current *.fla* file (and remove the `#include` directive).

Note that an `#include` directive is always executed, even when it is placed inside a conditional statement. For example the following code does not cause *theScript.as* to be left out of the movie:

```
if (false) {
  #include "theScript.as"
}
```

The `#include` directive commonly is used to incorporate the same block of code in multiple scripts or across Flash projects (much as you'd use an external asset library). You would do this in order to centralize your code, to maintain code a version-control system tool (such as CVS or Microsoft Visual Source Safe), or to use an external text editor that you pre

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

over the ActionScript editor. It is also handy when a programmer is working separately from, say, a graphic artist creating Flash animations. External files lend themselves well to code repositories, such as a library of classes or functions that are independent of the current timeline or movie clip. They tend to be less useful for code that needs to be integrated tightly w the Flash timeline.

To use international characters in an included file, save the file in UTF-8 format in your text editor, and add the following line to the beginning of the file (note the forward slashes and the space following the two dashes):

```
//!-- UTF8
```

The `#include` directive does not support UTF-16-encoded files.

Nested includes are supported, so a wrapper *.as* file can be used to include a series of related classes via a single statement much like a Java package.

The text in an included file is checked when a syntax check is performed using the Check Syntax command (Ctrl-T or Command-T) in the Actions panel menu (found under the arrow button in the upper-right corner of the panel).

Usage

Note that an `#include` directive begins with a pound sign (`#`), does not use parentheses, and must not end in a semicol Any `#include` statements that end in a semicolon will cause an error and will not successfully import the external scrip

Bugs

In Flash 5, a blank line is required between `#include` directives. The blank line can be inserted between the directives themselves or in the source of the file being included. This is a known bug that affects Flash 5 only. For example:

```
// This always works.
#include "file1.as"

#include "file2.as"

// In Flash 5, this only works if there's a blank line at the end of file1.a
#include "file1.as"
#include "file2.as"
```

In Flash 5 and Flash MX, if the last line of an included file is a comment, subsequently included files will not be incorporated (they will be commented out by the trailing comment in the previously included file). For example:

```
// WRONG!
doSomething();  // This comment will disable subsequently included files.
// RIGHT
// This comment is nice and safe.
doSomething();
```

Example

The following code imports an external *.as* file, named *myScript.as* , into the current *.fla* file. When using a relative path, *myScript.as* must be in the same folder as the *.fla* file containing the `#include` directive:

```
#include "myScript.as"
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

We can construct a relative path including a subdirectory. The following assumes that *myScript.as* is one level down from the current *.fla* file in a subdirectory named *includes* :

```
#include "includes/myScript.as"
```

Use two dots to indicate the folder above the current folder. The following assumes that *myScript.as* is one level *up* from t current *.fla* file:

```
#include "../myScript.as"
```

The following assumes that *myScript.as* is in a subdirectory named *includes* , adjacent to the subdirectory containing the current *.fla* file:

```
#include "../includes/myScript.as"
```

You can also specify an absolute path to any folder, such as:

```
#include "C:/WINDOWS/Desktop/myScript.as"
```

but absolute paths are not cross-platform and may need to be changed if you compile the *.fla* file on a different machine w a different directory structure. Note the differences in the drive letter specification and folder separators on the two platforms:

```
#include "C:/WINDOWS/Desktop/myScript.as"                  // Windows
#include "Mac HD:Desktop folder:working:myScript.as"       // Macintosh
```

See Also

"Externalizing ActionScript Code," in Chapter 16

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Chapter 18. ActionScript Language Reference

Infinity Global Property — Flash 5

a constant representing an infinite number — ***read-only***

```
Infinity
```

Description

A number that exceeds the maximum allowed numeric value in ActionScript is represented by the numeric constant `Infinity` (although incalculable numbers are represented by `NaN`). The largest value allowed in ActionScript is represented by `Number.MAX_VALUE`, which equates to 1.7976931348623e+308. Exceeding the largest allowed number in ActionScript is known as an *overflow condition* and is typically caused by a mathematical error.

Example

The result of a calculation that exceeds the largest allowed number is `Infinity`. For example:

```
Number.MAX_VALUE * 2;    // Yields Infinity
```

`Infinity` also results when dividing a positive number by zero:

```
1000 / 0;                // Yields Infinity
```

Usage

`Infinity` is shorthand for `Number.POSITIVE_INFINITY`.

See Also

`-Infinity`, *isFinite( )*, `NaN`, `Number.POSITIVE_INFINITY`; Section 4.3.3

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Chapter 18. ActionScript Language Reference

-Infinity Global Property — Flash 5

a constant representing an infinitely negative number — ***read-only***

```
-Infinity
```

Description

A number that exceeds the allowed *negative* numeric value in ActionScript is represented by the numeric constant `-Infinity` (although incalculable numbers are represented by `NaN`). The smallest negative value (the one with the largest absolute value) allowed in ActionScript is `-Number.MAX_VALUE`, which is equivalent to -1.79769313486231e+308. Exceeding the most-negative number representable in ActionScript is known as an *underflow condition* and is typically caused by a mathematical error.

Example

The result of a calculation that is smaller than (i.e., more negative than) the smallest allowed negative number is `-Infinity`. For example:

```
-Number.MAX_VALUE * 2;    // Yields -Infinity
```

`-Infinity` also results when dividing a negative number by zero:

```
-1000 / 0;                // Yields -Infinity
```

Usage

`-Infinity` is normally used in place of its more verbose equivalent `Number.NEGATIVE_INFINITY`.

See Also

`-Infinity`, *isFinite( )*, `NaN`, `Number.MAX_VALUE`, `Number.NEGATIVE_INFINITY`; "Special Values of the Number Datatype," in Chapter 4

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Chapter 18. ActionScript Language Reference

#initclip Pragma Flash 6

starts a movie clip initialization code block

```
#initclip order
```

Arguments

order

A nonnegative integer specifying the execution order of this `#initclip` block relative to all other `#initcli` blocks in the movie. Those with lower `order` numbers are executed before those with higher ones. Multiple `#initclip` blocks can share a single `order` position.

Description

The `#initclip` pragma starts a special block of code that allows you to perform initialization for a movie clip symbol before any instances of it are created. As explained in Chapter 14, `#initclip` typically is used to create so-called components, which are subclasses of the *MovieClip* class that have been made self-contained for easy use by nonprogram Therefore, the `#initclip` code block typically is used to register a movie clip symbol with an ActionScript class (i.e., create a subclass of the *MovieClip* class, as shown in Chapter 14). An `#initclip` block can appear only on frame 1 of movie clip symbol (it causes a compile-time error elsewhere). It is executed only once per movie, according to the export settings for the symbol (set under Options Linkage in the Library panel):

- If the symbol is not exported, the `#initclip` block is executed immediately before the frame where the first inst of the symbol appears.
- If the symbol is exported for ActionScript and "Export in first frame" is not selected, the `#initclip` block, as in previous case, is executed before the frame where the first instance of the symbol appears.
- If the symbol is exported for ActionScript and "Export in first frame" is selected, the `#initclip` block is execute before the first frame of the movie.

When the `#initclip` code block is executed, instances of the associated movie clip symbol (even those placed at auth time), haven't yet been created, so the code executes in the context of the main timeline (`_root`). Hence, `#initclip` has no access to any instance of the symbol in which it is defined. This behavior is intentional; `#initclip` is used to initialize the whole symbol, not a specific instance, so it must run before any instances are created.

In its most basic form, an `#initclip` block looks like this:

```
// CODE ON FRAME 1 OF A MOVIE CLIP SYMBOL
#initclip
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

```
// ...code to link the movie clip symbol to an ActionScript class
#endinitclip
```

The `#endinitclip` pragma closes the `#initclip` block (if `#endinitclip` is omitted, Flash generates a compi time error). The following code shows a more complete `#initclip` block placed in a movie clip symbol named *ballSy* The code creates a custom *MovieClip* subclass, *Ball* , and then it uses *Object.registerClass( )* to link the *ballSymbol* (in th movie's Library) to the ActionScript *Ball* class. Although the `#initclip` block executes only once, it typically is used define a constructor that will execute every time an instance of the symbol is created. See Chapter 14 for details on movie subclasses.

```
// CODE ON FRAME 1 OF THE MOVIE CLIP SYMBOL ballSymbol
#initclip

// Create the custom ActionScript class, Ball.
// The constructor is defined when #initclip executes. The constructor runs
// whenever a ballSymbol instance is placed on stage (dynamically or manuall
// with the keyword this pointing to that instance.
function Ball () {
  // Note that this.color is provided via attachMovie()'s initObj.
  // See MovieClip.attachMovie() for details.
  this.setRGB(this.color);
}

// Set the Ball superclass to MovieClip.
Ball.prototype = new MovieClip();

// Associate the Ball class with the library symbol named ballSymbol.
Object.registerClass("ballSymbol", Ball);

#endinitclip
```

When an instance of *ballSymbol* is placed on stage (manually or via ActionScript), the *Ball* constructor runs with `this` pointing to the instance. The instance inherits the properties and methods of both *Ball* and *MovieClip* . For example:

```
// Invoke the Ball constructor, which sets the redBall color. Note that
// the color property is assigned the value 0xFF0000 by the attachMovie()
// method's initObj parameter.
this.attachMovie("ballSymbol", "redBall", 1, { color: 0xFF0000 });
// Invoke a MovieClip method on the new instance.
redBall.play();
```

Interestingly, all the code in our `#initclip` example could have been placed on the main timeline. Creating a custom *MovieClip* subclass and linking it to a movie clip symbol is not unique to `#initclip` code. However, by placing the c directly in the movie clip symbol, we create a portable package that can easily be moved from one *.fla* file to another. To u our *MovieClip* subclass, a developer simply drags *ballSymbol* from our Library to hers and creates instances as usual. In f to make the movie clip even easier for nonprogrammers to use, we can define component parameters for it, allowing insta to be customized via the Properties inspector. For more information on component parameters, see Chapter 16 and Macromedia's component-building article at:

http://www.macromedia.com/support/flash/applications/creating_comps/

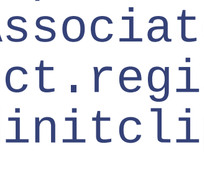
Returning to our example, we saw earlier that `#initclip` code executes in the context of the main timeline. Hence, the constructor function *Ball* is actually created on the main timeline and will appear as `_root.Ball` . If we know we will

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

create any subclasses of *Ball* , we can even delete the constructor from the main timeline by adding this code as the last lir our `#initclip` block:

```
delete Ball;
```



A constructor registered to a movie clip symbol is stored internally by the Flash Player and will execute normally even after being deleted. However, the `_root.Ball` function should not be deleted if it defines static properties (e.g., `Ball.gravity`) or if other movie clip symbols use it as a superclass. Furthermore, to be used as a superclass, the *Ball* function must be created before any of its subclasses.

To guarantee that a superclass, such as our *Ball* class, is created before its subclasses, we use the `order` parameter of th `#initclip` pragma, which assigns the code block an execution priority level. Multiple `#initclip` blocks can share same `order` level, but those with lower numbers are always executed before those with higher ones. Blocks without a specified `order` are executed before all numbered blocks. For example, to create a *BasketBall* subclass of *Ball* , we use

```
// CODE ON FRAME 1 OF THE MOVIE CLIP SYMBOL ballSymbol
// Note the order parameter!
#initclip 1
// Create the custom ActionScript class
function Ball () {
  this.setRGB(this.color);
}

// Set the Ball superclass to MovieClip
Ball.prototype = new MovieClip();
// Associate the Ball class with the library symbol named ballSymbol
Object.registerClass("ballSymbol", Ball);
#endinitclip

// CODE ON FRAME 1 OF THE MOVIE CLIP SYMBOL basketballSymbol
// Note the order parameter!
#initclip 2
// Create the custom ActionScript class
function BasketBall () {
  // Invoke the superclass constructor
  super();
}

// Set BasketBall's superclass to Ball
BasketBall.prototype = new Ball();
// Associate the BasketBall class with the Library symbol named basketballSy
Object.registerClass("basketballSymbol", BasketBall);
#endinitclip
```

Because functions defined in an `#initclip` block are scoped to the `_root` timeline, all variable, property, and method references inside the function must be qualified explicitly.

For example, to create a fully qualified reference, we must use `this._x` to refer to the current movie clip's `_x` property;

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

use an implicit reference to `_x` , as shown next, the property referenced is `_root._x` !

```
Ball.prototype.onEnterFrame = function () {
  // WRONG! Moves _root, NOT the ball!
  _x++;

  // RIGHT. Moves the ball.
  this._x++;
}
```

Duplicate class names inside separate `#initclip` blocks do not conflict. For example, if a class constructor named *Vel* is created inside two different `#initclip` blocks, each movie clip will be associated with its own version of the functic `_root.Vehicle` will point to the last one created. This prevents multiple components from breaking due to name con However, as a matter of good form, using identical class names in a single movie is not recommended. It's normally good to store each `#initclip` class on the `_global` object under a unique namespace, such as `_global.org.moock` described in Chapter 12 and shown throughout Chapter 14.

Code in an `#initclip` block is not limited to linking a class to a movie clip symbol, although that is its most common There's nothing stopping `#initclip` from performing some other task the first time a particular symbol is used in a mc The `#initclip` directive can define an arbitrary block of code that executes before the symbol containing the `#init` block is used for the first time (or when the movie starts, if the symbol is exported in the first frame of the movie):

For example, a movie clip exported for ActionScript in the first frame can be a convenient way to package a global library following code creates a *Timer* class that is available throughout a movie. The movie clip symbol is simply a container fo code; an instance of the symbol is never created.

```
// CODE ON FIRST FRAME OF AN EXPORTED MOVIE CLIP
#initclip
_global.Timer = function () {
  // ...Timer class code goes here...
};
#endinitclip

// CODE ON ANY TIMELINE IN THE MOVIE
gameclock = new Timer();
```

For much more information on building movie clip subclasses, see Chapter 14.

See Also

`#endinitclip` , *MovieClip.attachMovie( )* , *MovieClip.duplicateMovieClip* ( ), *Object.registerClass( )* ; Chapter 14

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Chapter 18. ActionScript Language Reference

instanceof Operator Flash 6

determines whether an object is an instance of a given class

```
object instanceof classConstructor
```

Operands

object

Any object.

classConstructor

A reference to any class's constructor function.

Returns

The Boolean `true` if *`object`* is an instance of *`classConstructor`*; otherwise, it returns `false`.

Usage

The *instanceof* operator does not take parentheses (it is an operator, not a function).

Example

```
var now = new Date();
trace(now instanceof Date);  // Displays: true
```

See Also

typeof; the *Constructor* subheading for the *Date* class; Chapter 12; Section 5.12.7

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Chapter 18. ActionScript Language Reference

int( ) Global Function

Flash 4; deprecated in Flash 5 in favor of analogous Math methods

---

truncate the decimal portion of a number

```
int(number)
```

Arguments

number

A number or an expression that yields a number, typically with a fractional (decimal) portion.

Returns

The integer portion of `number`.

Description

The *int( )* function was used in Flash 4 to extract the integer portion of a number. It effectively rounds positive numbers down and rounds negative numbers up. The *int( )* function works only for numbers in the range -2147483648 (-2^{31}) to 2147483647 ($2^{31}-1$); it produces `undefined` results for numbers outside this range. If `number` is a string composed of only numbers, *int( )* converts the string to a number before operating on it. If `number` is the Boolean value `true`, *int( )* returns the value 1. For all other nonnumeric data (including `undefined` and `null`), *int( )* returns the value 0.

Usage

The *int( )* function has been deprecated in favor of the more precise and standard *Math.floor( )*, *Math.ceil( )*, and *Math.round( )* methods. Use *parseInt( )* or *Number( )* to convert nonnumeric data to an integer or number.

Example

```
int(4.5)      // Yields 4
int(-4.5)     // Yields -4
int(3.999)    // Yields 3
```

The *int( )* function is useful to check if a number is a whole number, by comparing the original number to the result of the *int( )* function:

```
if (int(x) != x) {
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

```
  trace("Please enter a whole number for your age in years");
}
```

See Also

Math.ceil( ), Math.floor( ), Math.round( ), Number( ), parseFloat( ), parseInt( )

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

isFinite( ) Global Function — Flash 5

check if a number is less than Infinity and greater than -Infinity

```
isFinite(number)
```

Arguments

number

Any numeric value or expression that yields a numeric value.

Returns

The Boolean `true` if the number falls between `-Number.MAX_VALUE` and `Number.MAX_VALUE` (inclusive); otherwise, `false`. If *`number`* does not belong to the *number* datatype, *`number`* is converted to the *number* type before *isFinite( )* executes.

Description

The *isFinite( )* function simply checks if a number is in the legal numeric value range of ActionScript. It is equivalent to checking whether *`number`* is a number less than `Infinity` and greater than `-Infinity`. Use *isFinite( )* before executing code that requires a legitimate number to operate properly.

Example

```
if (!isFinite(x * y)) {           // Test if the number is not finite
  trace("The answer is too large to display. Try again.");
}
isFinite(-2342434);               // Yields true
isFinite(Math.PI);                // Yields true
isFinite(Infinity)                // Yields false
isFinite(-Infinity)               // Yields false
isFinite(NaN)                     // Yields false
```

See Also

`-Infinity`, `Infinity`, *isNaN( )*, `Number.MAX_VALUE`, `Number.MIN_VALUE`; Section 4.3.3

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Chapter 18. ActionScript Language Reference

isNaN( ) Global Function — Flash 5

equality test for the special NaN value

`isNaN(`*`value`*`)`

Arguments

value

The expression to test.

Returns

The Boolean `true` if *`value`* is the special numeric value `NaN` ; otherwise, `false` .

Description

To test whether or not a value is equal to the special numeric value `NaN` , we must use the *isNaN( )* function, because `NaN` test equal to itself in an ordinary equality test. For example, the expression:

```
NaN =  = NaN;
```

yields the value `false` . The *isNaN( )* function is often used to check whether a mathematical error (such as zero divide has occurred, or whether converting a value to a legitimate number has failed. Because *isNaN( )* returns `true` when the e is not a valid numeric expression, you'll often use the logical *NOT* operator (!) along with *isNaN( )* (i.e., something that is number *is* a number). Note that 0/0 yields `NaN` , but any positive number divided by 0 yields `Infinity` , and any neg number divided by 0 yields `-Infinity` .

Example

```
// Set a value
var x = "test123";
// Check if x is a legitimate number before using it is in a math expression
// is a handy for user input in text fields, which are always treated as str
if (!isNaN(parseFloat(x))) {
  var y = parseFloat(x) * 2;
}
```

See Also

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

isFinite( ) , `NaN` ; "Special Values of the Number Datatype," in Chapter 4

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Chapter 18. ActionScript Language Reference

Key Object — Flash 5

determine the state of keys on the keyboard

```
Key.property
Key.methodName()
```

Properties

Table 18-8 lists the properties of the *Key* object.

Table 18-8. Key object keycode properties

Property	Equivalent keycode	Property	Equivalent keycode
BACKSPACE	8	INSERT	45
CAPSLOCK	20	LEFT	37
CONTROL	17	PGDN	34
DELETEKEY	46	PGUP	33
DOWN	40	RIGHT	39
END	35	SHIFT	16
ENTER	13	SPACE	32
ESCAPE	27	TAB	9
HOME	36	UP	38

Methods

addListener*( )*

Registers an object to receive *onKeyUp( )* and *onKeyDown( )* events.

getAscii*( )*

Returns the ASCII value of the last key pressed.

getCode*( )*

Returns the keycode of the last key pressed.

isDown*( )*

Checks if a specific key is currently depressed.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

isToggled( )

Checks if the Num Lock, Caps Lock, or Scroll Lock keys are activated.

removeListener( )

Cancels event notices for the specified listener.

Listener Events

onKeyDown( )

Occurs when a key is depressed.

onKeyUp( )

Occurs when a key is released.

Description

The *Key* object is used to determine which keys are currently depressed and which key was last released. We can use it to build interfaces controlled by the keyboard, such as a game with a spaceship moved via the arrow keys.

Because not all keyboards are identical, keyboard-controlled interfaces can sometimes be tricky to create. By choosing our scripting tools correctly, however, we can ensure that all users have the same experience.

There are two general approaches to detecting keyboard commands:

- We can check if a key is currently depressed via the *isDown( )* method. This is recommended for cases in which the keyboard state needs constant *polling* (i.e., checking repeatedly), such as in video games. To poll the keyboard, use *isDown( )* to check the keyboard state within a *MovieClip.onEnterFrame( )* handler or during an interval created via *setInterval( )*.

- We can check which key was last released using the *getCode( )* and *getAscii( )* methods. This is recommended for typical keyboard-driven interfaces in which specific operations are performed in response to individual keys being pressed. We ordinarily use these methods from an *onKeyDown( )* or *onKeyUp( )* event listener, or a movie clip's *onClipEvent(keyDown)* or *onClipEvent(keyUp)* handler. There is no need to constantly check (i.e., poll) for the last key pressed. In fact, doing so leads to erroneously repeating some operation even if a key wasn't pressed repeatedly. That is, you should generally check *getCode( )* and *getAscii( )* from a listener or clip event only to guarantee the code is executed once and only once for each keystroke.

> The *isDown( )* method is synchronous, as is the *isToggled( )* method. They both report on the *current state* of the keyboard at the time the method is called. For example, *isDown( )* returns `false` if the key of interest is released before the call to *isDown( )* is made. On the other hand, the *getAscii( )* and *getCode( )* methods are asynchronous. They report on the last key pressed, even if it has been released. Therefore, it is usually a mistake to call *getAscii( )* or *getCode( )* unless a keyboard event has been detected via, say, an *onKeyDown( )* listener. Both *getAscii( )* and *getCode( )* will continue to report the same value indefinitely (or until another key is pressed), even if no key is currently being pressed.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

The so-called *Windows virtual keycode* (or simply *keycode*) returned by *getCode( )* and required by *isDown( )* is a number representing the physical keys on the keyboard, not the symbols on those keys and not their ASCII values. By using the keycode, we can identify keys even when a movie is running on different operating systems or when two keyboards use different languages or have different symbol layouts.

On most keyboards, the keycodes of the keys A to Z are the same as the code points (65-90) for the equivalent uppercase Latin 1 letters (also the same as their ASCII codes). The keycodes of the keys 0 to 9 are, likewise, the same as the Latin 1 and ASCII values for those numbers (48-57). The key codes of other keys do not necessarily match Latin 1 code points, although some, such as the Tab key, match the ASCII values. However, many of the nonletter and nonnumber keycodes are available as properties of *Key*, as shown in Table 18-8. For example, we don't have to remember that the up arrow uses keycode 38; we simply use the `Key.UP` property. The following code checks whether the up arrow key is currently depressed:

```
if (Key.isDown(Key.UP)) {
  trace("The up arrow is being pressed");
}
```

When working with a keycode that is not a letter or a number and is not available as a property of *Key*—such as those of the function keys (F1, F2, etc.)—it's safest to create a quick test movie to check the keycode of the desired key, as follows:

```
var keyTester = new Object();
keyTester.onKeyDown = function () {
  trace(Key.getCode());
}
Key.addListener(keyTester);
```

When testing keys in Test Movie mode, be sure to disable keyboard shortcuts via Control Disable Keyboard Shortcuts.

The keycodes are listed in Appendix B.

Usage

The *onKeyDown( )* and *onKeyUp( )* listener events were added to the *Key* object in Flash 6. They are preferred over *MovieClip.onKeyDown( )* and *MovieClip.onKeyUp( )* and the Flash 5-style *onClipEvent(keyDown)* and *onClipEvent(keyUp)* handlers in most cases.

However, the following caveats apply:

- In Flash 6, the *updateAfterEvent( )* function, which refreshes the screen between frames, can be used only with *MovieClip* mouse and key events and *setInterval( )*. Hence, to create instant-refresh keyboard applications, use *onClipEvent(keyDown)* and *onClipEvent(keyUp)*, which are directly analogous to *Key.onKeyDown( )* and *Key.onKeyUp( )*.

- To capture keystrokes while a specific movie clip has input focus only, use the *MovieClip.onKeyDown( )* and *MovieClip.onKeyUp( )* handlers. See those entries for details.

See Also

MovieClip.onKeyDown( ), *MovieClip.onKeyUp( )*; Chapter 10, Appendix B

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Chapter 18. ActionScript Language Reference

Key.addListener( ) Method — Flash 6

registers an object to receive onKeyDown( ) and onKeyUp( ) events

```
Key.addListener(listener)
```

Arguments

listener

An object with a defined *onKeyDown( )* and/or *onKeyUp( )* method.

Description

The *addListener( )* method adds `listener` to the list of objects notified when *Key*'s *onKeyDown( )* and *onKeyUp( )* events occur. The `listener` is an object of any class that defines methods by the name *onKeyDown* and/or *onKeyUp*; these need not be built-in methods of the object but can instead be custom-defined. The methods are invoked when the corresponding event occurs for *Key*. To stop a listener object from receiving events, use *Key.removeListener( )*.

Multiple objects can respond independently to each *Key* event. When multiple listeners are registered to *Key*, their methods are invoked in the order the listeners were added.

For general information on how listeners and callback functions work, see Chapter 10. For sample *Key* listener code, see *Key.getAscii( )* and *Key.onKeyDown( )*.

See Also

Key.getAscii( ), Key.onKeyDown( ), Key.onKeyUp( ), Key.removeListener( ); Chapter 10

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Key.getAscii( ) Method Flash 5

returns the ASCII value of the last key pressed

```
Key.getAscii()
```

Returns

An integer representing the ASCII value of the last key pressed.

Description

The *getAscii( )* method returns the ASCII value of the last key that was pressed (note that it does not accept any arguments). Since not all keys have an ASCII value, *getAscii( )* is normally used for detecting letters and numbers from the Latin 1 character set (used for Western European languages). Unlike *getCode( ), getAscii( )* distinguishes between upper- and lowercase letters. But unlike *getCode( ),* it cannot differentiate between two keys with the same ASCII value, such as the 8 key on the main keyboard and the 8 key on the numeric keypad.

To convert an ASCII value to a string character, use *String.fromCharCode( ),* as shown in the following example.

To detect the pressing of specific keys relative to their physical location on a keyboard rather than their ASCII value (as would be desirable when using four keys in a diamond pattern to control game play), use *getCode( ).* Both *getAscii( )* and *getCode( )* are typically used within an *onKeyDown( )* event listener, so that they are called once and only once per keypress.

Example

The following example shows how to distinguish between uppercase *A* and lowercase *a.*

```
if (Key.getAscii() =  = 65) {
  trace("Uppercase A was pressed last.");
} else if (Key.getAscii() =  = 97) {
  trace("Lowercase a was pressed last.");
} else {
  trace("Some other key was pressed last.");
}
```

Note that using *getCode( )* in this example would not allow you to distinguish between uppercase A and lowercase a, as both use keycode 65. To rewrite it with *getCode( )*, you'd have to check the state of the Shift key, with *isDown( ),* or the Caps Lock key, with *isToggled( ).*

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

The following example demonstrates keystroke detection for a simple hangman word game. It uses an *onKeyDown( )* event listener to detect a keypress and adds that key to a list of user guesses. Notice that *keyListener* is a generic object, demonstrating that any object defining an *onKeyDown( )* or *onKeyUp( )* callback handler can act as a listener. Objects without such callback handlers don't get notified of keyboard events, even if the objects are added to the event listener list with *Key.addListener( )*.

```
keyListener = new Object();
userGuesses = new Array();
guessNum = 0;
keyListener.onKeyDown = function () {
  var lastKey = Key.getAscii();
  userGuesses[guessNum] = String.fromCharCode(lastKey);
  guessNum++;
  trace(userGuesses);
}
Key.addListener(keyListener);
```

See Also

Key.getCode( ), *Key.isDown( )*; Appendix B

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Chapter 18. ActionScript Language Reference

Key.getCode( ) Method Flash 5

returns the keycode of the last key pressed

```
Key.getCode()
```

Returns

An integer representing the keycode of the last key pressed.

Description

The *getCode( )* method returns the keycode of the last key that was pressed; the keycode is an arbitrary number representing the physical location of a key on the keyboard. On non-Windows operating systems, the native keycode system is translated automatically by Flash to the Windows equivalent, so *getCode( )* provides a cross-platform means of detecting specific physical keys. The *getCode( )* method can also be used to differentiate between two keys with the same ASCII value. For example, it can differentiate between the 8 key on the main keyboard and the 8 key on the numeric keypad, whereas *getAscii( )* cannot. However, *getCode( )* cannot differentiate between upper- and lowercase letters (for example, *A* and *a* use the same keycode because they are produced using the same key). Nor does it detect modifier keys such as the Ctrl or Shift key.

Many common keycode values are available as properties of the *Key* object, as shown in Table 18-8 under the Description for the *Key* object. To determine the keycode of a particular key, see Appendix B, or construct a keycode tester as follows:

1. Create a new Flash document.
2. On frame 1, add the following code:

```
var keyTester = new Object();
keyTester.onKeyDown = function () {
  trace(Key.getCode());
}
Key.addListener(keyTester);
```

3. Choose Control Test Movie.
4. Choose Control Disable Keyboard Shortcuts.
5. Click the movie's Stage.
6. Press a key. The key code for that key will appear in the Output window.

Example

Unlike *isDown( )*, *getCode( )* is useful for creating interfaces in which an individual key press has a single,

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

direct result. Both *getAscii( )* and *getCode( )* are typically used with an *onKeyDown( )* event listener, so that they are called once and only once per keypress.

For example, the user may be allowed to skip a movie's intro by pressing the spacebar. When the spacebar is depressed, we send the playhead to the main interface of the movie (on the main timeline) as follows:

```
keyListener = new Object();
keyListener.onKeyDown = function () {
  if (Key.getCode() =  = Key.SPACE) {
    _root.gotoAndStop("mainInterface");
  }
}
Key.addListener(keyListener);
```

See Also

Button keyPress, *Key.getAscii( )*, *Key.isDown( )*; Appendix B

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Chapter 18. ActionScript Language Reference

Key.isDown( ) Method Flash 5

check whether a specific key is currently depressed

```
Key.isDown(keycode)
```

Arguments

keycode

A number representing the keycode of the key to check. Can also be one of the `Key` constants (e.g., `Key.UP` , `Key.BACKSPACE`).

Returns

A Boolean indicating whether the key specified by `keycode` is currently depressed (`true`) or not (`false`).

Description

The *isDown( )* method tells us whether the key specified by `keycode` is currently being pressed, such as:

```
trace(isDown(65));    // Displays: true if the "A" key is depressed during t
```

It offers immediate access to the state of the keyboard and is best used with systems that require constant key-based input detect the pressing of simultaneous keys.

The *isDown( )* method offers three advantages over *getCode( )* and *getAscii( )* ; it can:

- Detect the state of modifier keys such as Ctrl, Command, Shift, and Option, which do not trigger keyboard events o own. (To determine whether the Caps Lock, Num Lock, and Scroll Lock keys are in their locked (down) state, use *i)* instead.)
- Detect the state of arbitrary keys, allowing programmers to check for the simultaneous pressing of multiple keys.
- Detect whether a key is currently being pressed (*getCode( )* and *getAscii( )* tell you which key was last pressed, but have been subsequently released).

Unlike *getCode( )* and *getAscii( )* , which always return the last key pressed (even if it has since been released), *isDown( )* `keycode` argument specifying which keycode to test. By checking both `Key.UP` and `Key.RIGHT` in separate calls t *)* , for example, we can determine whether a spaceship in a game should be moved diagonally. Depending on the placeme specific keys being tested, the maximum number of keys that can be detected simultaneously may be as low as three.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Example

The *isDown( )* method normally is used to create systems that undergo repeated updates with each passing frame. In the fc code, we rotate and thrust a spaceship on any frame where the appropriate arrow keys are being pressed. Note that if you r detect two keys simultaneously, you should use separate *if* statements. In this example, the state of the right arrow key is i the left arrow key is depressed. Regardless, the state of the up arrow key is always checked in a separate *if* statement. A w version of this spaceship example is available at the online Code Depot:

```
// Code for a spaceship clip
spaceship_mc.onEnterFrame = function () {
  if (Key.isDown(Key.LEFT)) {          // Left arrow
    this._rotation -= 10;
  } else if (Key.isDown(Key.RIGHT)) {  // Right arrow
    this._rotation += 10;
  }
  if (Key.isDown(Key.UP)) {            // Up arrow
    this.thrust += 10;
  }
}
```

See Also

Button keyPress , *Key.getAscii( )* , *Key.getCode( )* , *Key.isToggled( )* ; Appendix B

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Key.isToggled( ) Method Flash 5

check whether the Caps Lock, Num Lock, or Scroll Lock keys are activated

```
Key.isToggled(keycode)
```

Arguments

keycode

An integer keycode, usually the keycode of the Caps Lock key (20), Num Lock key (144), or Scroll Lock key (145). Can also be the key constant `Key.CAPSLOCK`.

Returns

A Boolean indicating whether the key specified by *`keycode`* is locked on (`true`) or off (`false`).

Description

The *isToggled( )* method detects the state of the special Caps Lock, Num Lock, or Scroll Lock keys. Unlike other keys, these keys have a locked "on" state and an unlocked "off" state, indicating whether or not the feature they represent is active. The return value of *isToggled( )* tells us if the specified key is locked or not. Though *isToggled( )* accepts any *`keycode`*, the return value is useful only for special keys that support a toggle feature. For all other keycodes it returns `false`. To detect the state of other keys, use *isDown( )*. Note that *getCode( )* or *getAscii( )* detect the last key pressed, which may have been subsequently released, and therefore do not reflect the current state of a key.

See Also

Key.getAscii( ), Key.getCode( ), Key.isDown( ); Appendix B

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Chapter 18. ActionScript Language Reference

Key.onKeyDown( ) Listener Event Flash 6

occurs when a key is depressed

```
listener.onKeyDown()
```

Description

The *onKeyDown( )* and *onKeyUp( )* events are the keyboard analogues of *Mouse.onMouseDown( )* and *Mouse.onMouseUp( )*. Together, they provide fundamental tools for coding keyboard-based interactivity. The *onKeyDown( )* event can be handled by an event listener, as specified by `listener`, that has been added using *addListener( )*. The *onKeyDown( )* event occurs whenever a key on the keyboard is depressed, including modifier keys such as Shift and Ctrl. When a key is held down, *onKeyDown( )* may occur repeatedly, depending on the operating system and keyboard setup. Unlike the *on(keyPress)* button event, *onKeyDown( )* occurs when any key—not just a specified key—is pressed.

> An *onKeyUp( )* event handler is sometimes preferred to *onKeyDown( )* when inexperienced users, such as children, tend to hold down a key, potentially generating multiple unwanted *keyDown* events for each *keyUp* event. Conversely, an *onKeyDown( )* handler is generally more responsive than an *onKeyUp( )* handler and therefore should be used if responsiveness is important.

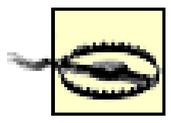
To *trap* (i.e., detect or *catch*) an *onKeyDown( )* event, we must first assign a callback function to the `onKeyDown` property of some object, such as `listener`, and then add `listener` as a listener using *Key.addListener( )*, as follows:

```
// Create a generic object
keyListener = new Object();

// Assign a function to the object's onKeyDown property
keyListener.onKeyDown = function () {
  trace("Some key was pressed!");
}

// Register the listener with the Key object
Key.addListener(keyListener);
```

Any number of listener objects of any class can respond to the *onKeyDown( )* event.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir



Keyboard events are triggered only if the Flash Player has mouse focus. Users must click the Stage of a movie before the movie's keystroke handlers will become active. Consider forcing users to click a button before entering any keyboard-controlled section of a movie. This ensures that the Flash Player has keyboard focus.

You'll notice that our preceding *onKeyDown( )* function does not tell us which key was pressed. If we're waiting for generic input, such as the user pressing any key to skip to the next section of a movie, we might not care which key was pressed. But often, we want to tie some action to a specific key. For example, we might want different keys to turn a spaceship in different directions.

Using *onKeyDown( )* handlers is appropriate for navigational interfaces that treat each keypress as a discrete action in need of a response (e.g., the spacebar causing a slide show presentation to advance). If you don't care which key was pressed, simply execute the desired code from your *onKeyDown( )* listener. To determine which key triggered the *onKeyDown* event, use the *Key.getAscii( )* or *Key.getCode( )* method inside a conditional (*if* statement).

For cases when you want to know the current state of the keyboard, or in applications such as games, that require instant, continuous feedback from potentially simultaneous keypresses, use the *Key.isDown( )* and *Key.isToggled( )* methods instead. Those methods would ordinarily be used within a handler that is called repeatedly, allowing you to check the state of the keyboard even if the system doesn't generate repeated events while keys are being held down. For example, you might check the keyboard state within a *MovieClip.onEnterFrame( )* handler, or using an interval created by *setInterval( )*.

The following example creates an *onKeyDown( )* listener that tells us the ASCII value of the last key pressed:

```
keyListener = new Object();
keyListener.onKeyDown = function () {
  lastKeyPressed = String.fromCharCode(Key.getAscii());
  trace("You pressed the '" + lastKeyPressed + "' key.");
}
Key.addListener(keyListener);
```

The following example checks whether the up arrow was the last key pressed:

```
keyListener = new Object();
keyListener.onKeyDown = function () {
  // Check to see if the up arrow was the last key pressed.
  // The up arrow is represented by the Key.UP property.
  if (Key.getCode() =  = Key.UP) {
    trace("The up arrow was the last key depressed");
  }
}
Key.addListener(keyListener);
```

As a rule of thumb, use *Key.getAscii( )* when you care about the *character* of the key that was pressed, such as whether it was an uppercase *A* or lowercase *a*. Use *Key.getCode( )* when your care which physical key was pressed, such as when using four adjacent keys for navigation, regardless of the characters they represent. See the Discussion under the *Key* object entry for more information.

The *onKeyDown( )* event executes immediately when a key is pressed, independent of the code on a movie's timeline(s). However, visual changes to the movie made by the listener are not rendered until the next frame refresh, which is determined by the frame rate of the movie. Unfortunately, in Flash 6, *updateAfterEvent( )* cannot be used to force a screen refresh from an *onKeyDown( )* listener; *updateAfterEvent( )* has an effect only when

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

invoked either from a movie clip mouse or key event, or a *setInterval( )* callback.

Handling special keys

To disable the Standalone Player's contextual menu commands (Open, Close, Fullscreen, etc.), add the following line of code to the beginning of your movie:

```
fscommand("trapallkeys", "true");
```

This command also prevents the Escape key from exiting fullscreen mode in a Standalone Player. To capture Escape in a Standalone Player, use:

```
keyListener.onKeyDown = function () {
  if (Key.getCode() =  = Key.ESCAPE) {
    // Respond to Escape keypress
  }
}
```

Note that the Escape key cannot be trapped in every browser. Furthermore, there is no way to disable the Alt key or the Windows Alt-Tab or Ctrl-Alt-Delete key sequences.

The following code captures Tab keypresses:

```
keyListener.onKeyDown = function () {
  if (Key.getCode() =  = Key.TAB) {
    trace("Tab was the last key depressed");
  }
}
```

In some browsers, Flash Player 5 detects the Tab key only within an *on(keyPress)* button event. The following code successfully detects the Tab key in all browsers using Flash Player 5:

```
// CODE ON BUTTON ON MAIN TIMELINE
on (keyPress "<Tab>") {
  // Set a dummy variable here
  foo = 0;
}

// CODE ON MOVIE CLIP ON MAIN TIMELINE
// Notice that this is a keyUp event. The keyDown event doesn't work with
// Flash 5 running in Internet Explorer.
onClipEvent (keyUp) {
  if (Key.getCode() =  = Key.TAB) {
    // Now place the cursor in myTextField on _level0
    Selection.setFocus("_level0.myTextField");
  }
}
```

In Flash Player 6, the Tab key can safely be detected with a *Key* listener. We typically trap the Tab key in order to move the insertion point to a particular text field in a form. However, as of Flash 6, custom Tab key behavior can be controlled precisely via the `tabIndex` and `tabEnabled` properties of the *TextField*, *MovieClip*, and *Button* classes, without the need for any *Key* listeners. For a Flash 5-compatible Tab order example, see:

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

http://moock.org/asdg/codedepot/interface-widgets/taborder.fla

To capture a key combination, such as Ctrl-G, use:

```
keyListener.onKeyDown = function () {
  if (Key.isDown(Key.CONTROL) && Key.isDown(71)) {
    // Respond to Ctrl-G. Keycode 71 is the keycode for the G key.
  }
}
```

To capture the Enter (or Return) key, use:

```
keyListener.onKeyDown = function () {
  if (Key.getCode() =  = Key.ENTER) {
    // Respond to Enter keypress (e.g., submit a form)
  }
}
```

Usage

To emulate *onKeyDown( )* in Flash 5, use a movie clip *onClipEvent(keyDown)* clip event attached to the clip at authoring time. For example:

```
onClipEvent (keyDown) {
  trace("Some key was pressed");
}
```

See Also

Key.addListener( ), Key.getAscii( ), Key.getCode( ), Key.isToggled( ), Key.onKeyUp( ), MovieClip.onKeyDown( )

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Key.onKeyUp( ) Listener Event — Flash 6

occurs when a key is released

```
listener.onKeyUp()
```

Description

The *onKeyUp( )* event is triggered when a key is depressed and then released. It is sometimes used in game programming to turn off something that was turned on by an earlier *onKeyDown( )* event—the classic example being a spaceship's thrust. The *onKeyUp( )* event can be handled by an event listener, as specified by `listener`, that has been added using *addListener( )*. The *onKeyUp( )* event can also allow a key to modify an application's behavior temporarily. For example, in the Flash authoring tool, holding down the spacebar temporarily switches to the Hand tool, and releasing the spacebar restores the previous tool. A similar approach can be used to show and hide things in an application, such as temporary menus.

To respond to an *onKeyUp( )* event, we must first assign a callback function to the `onKeyUp` property of some object, such as `listener`, and then add `listener` as a listener using *Key.addListener( )*, as follows:

```
// Make a generic object
keyListener = new Object();

// Assign a function to the object's onKeyUp property
keyListener.onKeyUp = function () {
  trace("A key was released!");
}

// Register the listener with the Key object
Key.addListener(keyListener);
```

Any number of listener objects can respond to the *onKeyUp( )* event. As with *onKeyDown( )*, in order to obtain useful information from an *onKeyUp( )* event, we normally use it with other methods of the *Key* object:

```
keyListener.onKeyUp = function () {
    trace("The last key released was " + Key.getAscii());
  }
}
```

The *onKeyDown( )* and *onKeyUp( )* events are triggered precisely once for each keypress and key release. When using *onKeyUp( )* and *onKeyDown( )*, you need not concern yourself with whether the key is still being pressed at any given instant. Using these events allows you to detect keypresses accurately even if the user releases the key between frames, and it also prevents you from checking the same key twice if it was pressed only once. In any case, you will ordinarily use the *Key.getCode( )* and *Key.getAscii( )* methods to check for the

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

last key pressed within an *onKeyDown( )* or *onKeyUp( )* listener.

The *onKeyUp( )* event executes immediately when a key is released, independent of the code on a movie's timeline(s). However, visual changes to the movie made by the listener are not rendered until the next frame refresh, which is determined by the frame rate of the movie. Unfortunately, in Flash 6, *updateAfterEvent( )* cannot be used to force a screen refresh from *onKeyUp( )*; *updateAfterEvent( )* has an effect only when invoked either from a movie clip mouse or key event, or a *setInterval( )* callback.

Usage

To emulate *onKeyUp( )* in Flash 5, use the *MovieClip* class's *on(keyUp)* clip event attached to a movie clip at authoring time. For example:

```
onClipEvent (keyUp) {
  trace("Some key was released");
}
```

See Also

Key.addListener( ), *Key.onKeyDown( )*, *MovieClip.onKeyUp( )*

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Key.removeListener( ) Method Flash 6

cancels event notices for the specified listener

```
Key.removeListener(listener)
```

Arguments

listener

An object that was previously registered as a listener for *Key* using *Key.addListener( )*

Returns

A Boolean; `true` if `listener` was found and removed, `false` if `listener` was not registered as an event listener for *Key*. The return value is useful for debugging code that removes listeners.

Description

The *removeListener( )* method stops `listener` from receiving *Key* event notifications. It should be used only after `listener` has been registered as an event listener via *addListener( )*. The listener object is not deleted but can be removed and added repeatedly.

Example

Event listeners should be removed when the task they perform is no longer required. For example, when a modal dialog box is displayed in a keyboard-controlled application, the application should ignore general keyboard input until the dialog box is closed. We can disable keyboard input outside the dialog box by removing the default key listener for the application, as shown here:

```
// Method definition for launching a dialog box from theApp
theApp.displayModalDialog = function () {
  // Disable the regular key listener
  Key.removeListener(this.defaultKeyListener);
  // Enable the temporary modal dialog key listener
  Key.addListener(this.modalKeyListener);
  // Continue with launching the dialog...
}
```

See Also

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Key.addListener( ), *Key.onKeyDown( )*, *Key.onKeyUp( )*; Chapter 10

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Chapter 18. ActionScript Language Reference

_leveln Global Property

a document level in the PlayerFlash 3 ***read-only***

```
_level0
_level1
_level2
...
_leveln
```

Description

Multiple *.swf* files can be loaded into the Flash Player for simultaneous display. Each loaded *.swf* can be placed in a movie clip or on its own "level" in the *document level stack* . The `_level`*n* property stores a reference to the main timeline of a *.swf* loaded into a document level in the Player. Each document level is represented by a numbered property, such as `_level0` , `_level1` , `_level2` , and so on. A *.swf* file on a higher level will obscure lower levels if they occupy the same portion of the Stage. The original document loaded into any Flash Player is considered `_level0` .

Note that `_level`*n* is not always synonymous with `_root` ; `_root` refers to the main timeline of the current movie, while `_level`*n* refers to the main timeline of a movie on a specific document level. To create global variables that are available from any document level, use `_global` .

Example

A `_level`*n* reference normally is used to control movies on other levels of the document stack. For example, here we play the movie on level 2:

```
_level2.play();
```

We can also use `_level`*n* in combination with movie clip references to control clips contained by a movie on any level in the document stack. For example:

```
_level1.orderForm.titleBar.play();
```

A `_level`*n* reference may also be used as the value of the *`target`* argument of several functions, including *loadMovie( )* , *unloadMovie( )* , *loadVariables( )* and *print( )* . If the level does not yet exist, you should specify the level reference within quotes. If used without quotes, a nonexistent level is considered `undefined` and may cause the command to operate on the current timeline instead of the new, undefined level. For example, when executed from the main timeline of `_level0` , the following will replace the movie in `_level0` if `_level1` has not yet been defined:

```
loadMovie("myMovie.swf", _level1);
```

The following is a safer approach if you can't guarantee that the level already exists:

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

```
loadMovie("myMovie.swf", "_level1");  // Works even if _level1 doesn't exist
```

However, the existence of `_level0` is always guaranteed, so we can always refer to it using `_level0` without quotes, as in:

```
startDrag(_level0, true);
```

See Also

`_global` , *loadMovie( )* , `_root` , *unloadMovie( )* ; "Importing External Movies," and "Movie and Instance Stacking Order," in Chapter 13

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

loadMovie( ) Global Function

Flash 4; Flash 6 adds .jpg loading and security restrictions

load an external .swf or .jpg file into the Player

```
loadMovie(url, target)
loadMovie(url, target, method)
```

Arguments

url

A string specifying the absolute or relative file path of the external *.swf* movie or *.jpg* image file to load. All URLs use forward slashes, and absolute URLs must include the protocol reference (e.g., *http://* or *file:///*). Relative URLs resolved relative to the URL of the movie on `_level0` (i.e., `_level0._url`) and use the *http://* protocol by default.

target

A string indicating the movie clip or document level that will host the external *.swf* or *.jpg* file. Can also be a refere to an existing movie clip or document level (references are converted to paths when used in a string context).

method

An optional string indicating the method by which to send variables to an external script. The legal values for `method` are `"GET"` and `"POST"` . This parameter must be a literal, not a variable or other expression. On Windows, prior Flash 6, the Standalone Player always used the `"GET"` method, regardless of the `method` specified. As of Flash both `"GET"` and `"POST"` are supported. See Appendix F.

Description

The *loadMovie( )* function is the global analogue of the *MovieClip.loadMovie( )* method. Both versions import the *.swf* or *.jpg* (JPEG) image file located at `url` into the Flash Player, but the global version uses `target` to identify the new location of the loaded file, whereas the *MovieClip* method version always uses the calling movie clip as the target. If `url` JPEG image, it cannot be a *.jpg* file saved in progressive format.

If `target` is a reference to an existing movie clip or a string indicating the path to a movie clip, the loaded *.swf* or *.jpg* placed into the specified clip (causing the eviction of any previous content). To load a movie or image into the current mo clip, use the empty string as the `target` parameter, as in:

```
loadMovie("myMovie.swf", "");
```

or use the *MovieClip.loadMovie( )* version with the keyword `this` :

```
this.loadMovie("myMovie.swf");
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

If *target* is a reference to an existing document level (such as _level2) or a string indicating the path to a docume level (such as "_level2"), then the *.swf* or *.jpg* file is placed into the specified document level. Loading a movie into _level0 clears the Player of all content on all levels and places the new *.swf* or *.jpg* file into _level0 .

Note that *loadMovie( )* executes asynchronously; the load operation does not actually begin until after all scripts on the cu frame execute (specifically, Flash requests the external asset from the operating system just before the screen is updated). the *.swf* or *.jpg* file begins loading into the Player, download progress can be monitored via *MovieClip.getBytesLoaded( )* *MovieClip* ._framesloaded . For details on preloading movies, see those entries and the following article:

http://design.oreilly.com/news/action_0501.html

Be careful not to check load progress before *loadMovie( )* actually executes. To ensure an accurate progress reading for a loading *.swf* file, set a flag on its first frame, indicating that loading has started:

```
var loadStarted = true;
```

Then, poll for that variable from outside the *.swf* file before checking load progress:

```
if (target_mc.loadStarted =  = true) {
  // It's safe to display accurate load progress...
}
```

It is possible to send variables along with a *loadMovie( )* invocation, in which case *url* normally is the location of a scri that returns a *.swf* or *.jpg* file based on the variables sent. To send variables with a *loadMovie( )* call, we include the *metl* argument (set to either "GET" or "POST"). "GET" sends the current movie clip timeline variables as a query string attached to the script *url* . "POST" sends the current movie clip timeline variables after the HTTP POST-request head Prior to Flash 6, the "POST" method was not available in the Standalone Player, but it is functional in the Flash 6 Standalone Player. Because most web servers restrict the length of URLs to no more than 1024 characters (and some to as little as 255 characters), use "POST" instead of "GET" to transfer larger amounts of data.

A *loadMovie( )* invocation that uses the "GET" method can pass variables to a loaded movie without the help of an intervening script, provided the movie is served from a web server and not by the local filesystem. For example, the follov code loads the external movie *myMovie.swf* into level 1 of the Player document stack, passing it the variables from the current timeline:

```
loadMovie("myMovie.swf", "_level1", "GET");
```

Variables passed to the loaded movie are defined on its main timeline. This technique works only when the *loadMovie( )* request is handled by a web server. Attempts to use the "GET" method with *loadMovie( )* using local files will cause an ' opening URL" error.

As of Flash 6, a movie from one domain can load a movie from another domain but cannot access its variables, properties functions unless it provides special permission. For full details, see *System.security.allowDomain( )* .

Usage

Be careful when using movie clip and level references as the *target* argument of *loadMovie( )* . If a *loadMovie( )* 's *target* argument yields undefined or the empty string, the *loadMovie( )* function uses the current timeline as its *target* (i.e., it replaces the current timeline with the newly loaded content). In particular, this causes problems for loac movies onto new, unoccupied levels. Consider the following code:

```
loadMovie("myMovie.swf", _level1);
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

If no `_level1` object exists prior to the execution of this statement, the code will load *myMovie.swf* into the timeline th contains the *loadMovie( )* statement, not `_level1` ! To avoid the problem, you can use *loadMovieNum( )* instead. Alternatively, you can use a string for the `target` parameter to *loadMovie( )* , as in:

```
loadMovie("myMovie.swf", "_level1");
```

In this case, the level will be created if it doesn't already exist (only `_level0` exists by default in all movies). For more information, see Section 13.7.

The global *loadMovie( )* function replaces the target path version of Flash 4's *Load Movie* action.

Example

Here are some *loadMovie( )* examples:

```
loadMovie("myMovie.swf", "_level1");     // Place myMovie.swf on level 1
loadMovie("myMovie.swf", "_level0");     // Place myMovie.swf on level 0
loadMovie("myMovie.swf", "myClip");      // Place myMovie.swf into myClip
// Replace the contents of the Player with coolmovie.swf, using an absolute
loadMovie("http://www.yourflashsite.com/coolmovie.swf", "_level0");
// Load a movie into level 1 from the Windows desktop. Note the
// file:/// protocol and the forward slashes.
loadMovie("file:///C|/WINDOWS/Desktop/animation.swf", "_level1");
// Load a JPEG image into imageHolder_mc
loadMovie("photo.jpg", "_level0.imageHolder_mc");
```

The following code loads a *.jpg* file into a movie for use as a button. It first creates a clip to act as the button, then creates inside that clip to act as the image holder. Note the clip nesting. If *onRelease( )* were attached to the image holder clip rat than the outer button clip, it would be deleted by the loading *.jpg* file.

```
this.createEmptyMovieClip("btn_mc", 1);
this.btn_mc.createEmptyMovieClip("imgHolder_mc", 1);
this.btn_mc.onRelease = function () {
  trace("Image clicked");
}
this.btn_mc.imgHolder_mc.loadMovie("buttonPhoto.jpg");
```

A generic image loading utility with pan, zoom, and preloading support is available at the online Code Depot.

See Also

loadMovieNum( ) , *MovieClip.loadMovie( )* , *System.security.allowDomain( )* , *unloadMovie( )* ; Section 13.3.4

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

loadMovieNum( ) Global Function

Flash 3; Flash 6 adds .jpg loading and security restrictions

---

load an external .swf or .jpg file into a document level

```
loadMovieNum(url, level)
loadMovieNum(url, level, method)
```

Arguments

url

A string specifying the absolute or relative file path of the external *.swf* movie or *.jpg* image file to load. All URLs must use forward slashes, and absolute URLs must include the protocol reference (e.g., *http://* or *file:///*). Relative URLs are resolved relative to the URL of the movie on `_level0` (i.e., `_level0._url`) and use the *http://* protocol by default.

level

A nonnegative integer, or an expression that yields one, indicating the number of the document level that will host the external *.swf* or *.jpg* file.

method

An optional string indicating the method by which to send variables to an external script. The legal values for *`method`* are `"GET"` and `"POST"`. This parameter must be a literal, not a variable or other expression. On Windows, prior to Flash 6, the Standalone Player always used the `"GET"` method, regardless of the method specified. As of Flash 6, both `"GET"` and `"POST"` are supported. See Appendix F.

Description

The *loadMovieNum( )* function is nearly identical to *loadMovie( )* used with a document level as its *`target`*, except that the target *`level`* is specified as a number rather than as a string. This means that *loadMovieNum( )* can load movies only into document levels, not host clips. The *loadMovieNum( )* function is safer than *loadMovie( )*, insofar as there is no chance of inadvertently replacing the current clip or movie; if the specified level doesn't exist, it will be created. If the specified level does exist, its occupant is replaced by the new *.swf* or *.jpg* file. It is valid to load a movie into `_level2` even if `_level1` hasn't been created.

The *loadMovieNum( )* function can be used when we wish to assign the level of a loaded movie or image dynamically, as in:

```
var x = 3;
loadMovieNum("myMovie.swf", x);
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

which could also be achieved via a string concatenation expression with the regular *loadMovie( )* function:

```
loadMovie("myMovie.swf", "_level" + x);
```

Usage

The global *loadMovieNum( )* function replaces Flash 3's *Load Movie* action, which accepted level numbers only (Flash 4 added the `method` parameter).

See Also

loadMovie( ), *MovieClip.loadMovie( )*

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Chapter 18. ActionScript Language Reference

loadVariables( ) Global Function — Flash 4

retrieve an external set of variables

```
loadVariables(url, target)
loadVariables(url, target, method)
```

Arguments

url

A string specifying the path to a variable source—either a server-side script that returns variables or a text file containing variables.

target

A string indicating the path to the movie clip or document level on which the loaded variables will be defined. Can also be a reference to a movie clip or document level (references are converted to paths when used in a string context).

method

An optional string indicating the method by which to send variables to an external script. If specified, the variables from the current timeline are sent to the script and `target` receives the loaded variables. If omitted, variables are retrieved but none are sent. The legal values for `method` are `"GET"` and `"POST"`. This parameter must be a literal, not a variable or other expression. On Windows, prior to Flash 6, the Standalone Player always used the `"GET"` method, regardless of the `method` specified. As of Flash 6, both `"GET"` and `"POST"` are supported. See Appendix F.

Description

Using *loadVariables( )*, we can import variables into a movie from a text file or a server-side application such as a Perl script. However, as of Flash 6, the *LoadVars* class provides a more flexible and powerful means of importing and exporting variables.

Variables loaded via *loadVariables( )* are scoped to the movie clip or level specified by `target` and are always of the *string* datatype. To attach loaded variables to the current timeline, use the empty string as the value of the `target` argument. For example:

```
loadVariables("myVars.txt", "");  // Loads variables from myVars.txt
                                  // onto the current timeline
```

Whether the variables to be loaded reside in a text file or are composed by a script, they must be formatted according to the rules of URL encoding, described in the *LoadVars* class entry.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

A text file for use with *loadVariables( )* is simply a plain text file containing URL-encoded variables (as described in the *LoadVars* class entry). To load variables from an external text file, we specify the path of the file as the `url` argument in our *loadVariables( )* function invocation. For example:

```
// Load the variables from myVariables.txt into the main movie timeline
loadVariables("myVariables.txt", "_root");
```

The *loadVariables( )* function can also be used with a script or server application that outputs URL-encoded variables. When a script sends data to a Flash movie in response to a *loadVariables( )* invocation, the script should set the MIME type of the data as: `"application/x-www-form-urlencoded"`. Here's a line from a typical Perl script to set the MIME type:

```
print "Content-type: application/x-www-form-urlencoded\n\n";
```

Though the name *loadVariables( )* suggests only a single direction of variable transmission, this function can also be used to *send* variables to a server-side script. To send all the variables defined on the current timeline to a script, we set the `method` argument of a *loadVariables( )* function invocation to either `"GET"` or `"POST"`. Variables are sent in URL-encoded format. If `method` is set to `"GET"`, the variables are appended as a query string to `url` and sent to the remote script. If `method` is set to `"POST"`, the variables are sent after the HTTP POST-request header. The `"POST"` method was not available in the Standalone Player for Windows prior to Flash 6, but it is functional in the Flash 6 Standalone Player for Windows (see Appendix F for details on support for GET and POST). Because most web servers restrict the length of URLs to no more than 1024 characters (and some to as little as 255 characters), use `"POST"` instead of `"GET"` to transfer larger amounts of data.

For important information on domain-based security restrictions that apply to *loadVariables( )*, see `System.security`.

Usage

The results of multiple *loadVariable( )* calls to the same `url` may be cached by some browsers, preventing new data from loading from the server. To avoid this problem, append a dummy variable to each *loadVariables( )* call so that the URL is unique. For example, here we generate a unique URL by appending the time in milliseconds:

```
loadVariables("http://www.mysite.com/cgi-bin/myScript.pl?cacheKiller="
              + new Date().getTime(), serverResponse);
```

Bugs

The POST method is not supported by Internet Explorer 4.5 on Macintosh. This problem was fixed in Version 5 of the browser.

On Macintosh, the *loadVariables( )* method does not work with Flash 5 executable Projectors (i.e., movies bundled with the Flash 5 Standalone Player into a single executable file). However, *loadVariables( )* does work with movies opened manually in the Standalone Player. To work around this problem, send data via the *XML.send( )* and *XML.sendAndLoad( )* methods.

See Also

loadVariablesNum( ), the *LoadVars* class, *MovieClip.loadVariables( )*, *System.security.allowDomain( )*; Chapter 17

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

loadVariablesNum( ) Global Function Flash 5

attach an external set of variables to a document level

```
loadVariablesNum(url, level)
loadVariablesNum(url, level, method)
```

Arguments

url

A string specifying the path to a variable source—either a server-side script that returns variables or a text file containing variables.

level

A nonnegative integer, or an expression that yields one, indicating the number of the document level on which the loaded variables will be defined.

method

An optional string indicating the method by which to send variables to an external script. If specified, the variables from the current timeline are sent to the script and `level` receives the loaded variables. If omitted, variables are retrieved but none are loaded. The legal values for `method` are `"GET"` and `"POST"`. This parameter must be a literal, not a variable or other expression. On Windows, prior to Flash 6, the Standalone Player always used the `"GET"` method, regardless of the method specified. As of Flash 6, both `"GET"` and `"POST"` are supported. See Appendix F.

Description

The *loadVariablesNum( )* function is nearly identical to *loadVariables( )*, except that it requires the target `level` to be specified as a number rather than as a string. This means that *loadVariablesNum( )* can attach variables to document levels only, not movie clips. The target `level` can be specified dynamically, as in:

```
var myLevel = 2;
loadVariablesNum("myVars.txt", myLevel);
```

A similar effect can be achieved using string concatenation with the regular *loadVariables( )* function:

```
loadVariables("myVars.txt", "_level" + myLevel);
```

See Also

loadVariables( )

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

LoadVars Class

Flash 6

---

export variables to, or import variables from, an external source

Constructor

`new LoadVars()`

Properties

contentType

The MIME content type used for *send( )* and *sendAndLoad( )* operations.

loaded

Status of a *load( )* or *sendAndLoad( )* operation.

Methods

decode*( )*

Convert a URL-encoded string of variables to properties.

getBytesLoaded*( )*

Check what portion of loading variables has arrived, in bytes.

getBytesTotal*( )*

Check the expected total byte size of loading variables.

load*( )*

Import URL-encoded variables from a text file or a server-side application.

send*( )*

Transfer URL-encoded variables to a server-side application or script.

sendAndLoad*( )*

Transfer variables to a server-side application, and receive variables in return.

toString*( )*

Returns the URL-encoded string of variables that would be sent by *send( )* or *sendAndLoad( )*.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Event Handlers

onData( )

Handler executed when external variable data finishes loading.

onLoad( )

Handler executed when external variables have been converted into object properties.

Description

The *LoadVars* class transfers variables to or from an external source, such as a text file or a server-side script. Table 18-9 shows typical *LoadVars* usage scenarios and lists the method associated with each scenario.

Table 18-9. LoadVars usage scenarios

Desired Action	Method
Retrieve external variables from a text file or server-side script.	*LoadVars.load( )*
Send ActionScript variables to a server-side script, and receive optional results in a web browser.	*LoadVars.send( )*
Send ActionScript variables to a server-side script, and receive results in Flash.	*LoadVars.sendAndLoad( )*

The *send( )*, *load( )*, and *sendAndLoad( )* methods must be invoked through an instance of *LoadVars*, not through the class itself. The properties of the instance are either the variables to be sent or those that have been received. The *send( )* method converts the properties of the *LoadVars* instance to a string of URL-encoded variables, and sends them as an HTTP request to a server-side application. The *load( )* method retrieves URL-encoded variables and converts them to properties of the *LoadVars* object. The *sendAndLoad( )* method does both. For details and examples, see each method's listing.

The *LoadVars* class provides the same basic service as the global *loadVariables( )* function, first introduced in Flash 4, but in a more logical format and with several important advantages:

- An *onLoad( )* handler executes when variables arrive, notifying you automatically rather than requiring you to poll for results as you must with *loadVariables( )*.
- Variable load progress can be monitored with *getBytesLoaded( )* and *getBytesTotal( )*, allowing you to diagnose problems or provide a status bar during lengthy operations.
- Variables are stored as properties of *LoadVars* objects, not in physical movie clips, allowing you to separate code from media more effectively. This also allows you to send a subset of variables from a timeline, rather than sending all variables on a timeline, as is the case with *loadVariables( )*.

Variables transferred by a *load( )* or *sendAndLoad( )* operation are formatted according to the rules of URL encoding, as follows:

- Every variable name is separated from its value with an equals sign, without spaces, as in `firstName=stephen`.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

- Multiple variable name/value pairs are separated by ampersands (`&`), as in `firstName=stephen&lastName=burke`.
- Spaces are replaced with plus (`+`) signs, not `%20`.
- Any character that is not a space, a number (1-9), or an unaccented Latin 1 letter (a-z or A-Z) is replaced as follows:
    - If `System.useCodepage` is `false` (the default), the character is replaced by one or more hexadecimal escape sequences of the form `%xx`, representing the UTF-8 sequence for the character. Characters in the Unicode range 0-127 are encoded as `%xx` (single-byte), characters 128-2047 are encoded as `%xx%xx` (double-byte), and characters 2048 and above are encoded as `%xx%xx%xx` (triple-byte).
    - If `System.useCodepage` is `true` and the system is not an Asian language double-byte system, the character is replaced with a hexadecimal escape sequence `%xx`, where `xx` is the hex code point for the character. The code point is determined using the system code page (typically Latin 1 on most Western computers). Note that on Macs, the system code page is Mac Roman, but Flash uses Latin 1 instead.
    - If `System.useCodepage` is `true` and the system is an Asian language double-byte system, the character is replaced with a hexadecimal escape sequence `%xx` or `%xx%xx`, using the double-byte character encoding supported by the system (for example, Shift-JIS).
- Flash 5 and previous Players behave as if `System.useCodepage` is true.

The following code shows the contents of a text file to be imported into Flash. The imported variables are `name` and `address`, which have the values `"stephen"` and `"65 nowhere st!"`, respectively (note that the exclamation point is encoded as `%21`):

```
name=stephen&address=65+nowhere+st%21
```

Generally, server-side applications automatically URL encode variables for transmission to Flash, but variables in text files must be URL-encoded by hand. For example, in Flash Player 5 (or when `System.useCodePage` is `true` in Flash Player 6), the high-ASCII character *e* is at code point 233, which is hex E9, so it is written manually as `%E9`. The word *resume* becomes `resum%E9`. In contrast, in Flash Player 6, when `System.useCodePage` is `false` (the default), the character *e* should be entered directly, and the variables file should be saved in Unicode UTF-8 format. (In Windows Notepad, the UTF-8 format can be chosen in the "Save as..." dialog box.) If the file cannot be saved as UTF-8, or if the character cannot be entered directly, the single-, double-, or triple-byte UTF-8 escape sequence for the character must be used. For example, the UTF-8 sequence for *e* is `%C3%A9`, so the word *resume* would be written as `resum%C3%A9`. You can check the UTF-8 escape sequence for a character in Flash Player 6 using the *escape( )* function. For example:

```
trace(escape("e"));  // Displays: %C3%A9
```

As a general rule, where many international characters are required, data should be loaded from Unicode-formatted XML files rather than via *LoadVars*.

For security reasons, *LoadVars* works only with hosts in the domain from which the movie was downloaded. For more information, see `System.security`.

The *LoadVars* class is not always the best means of importing data into Flash. For variables that need to be

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

imported only once at load-time in a browser, consider using the `FLASHVARS` attribute of the HTML `<OBJECT>` and `<EMBED>` tags (see Section 2.6). For large quantities of data, consider using the *XML* class.

See Also

loadVariables( ), `System.security`, the *XML* class; "Loading External Variables," in Chapter 2; Chapter 17

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Chapter 18. ActionScript Language Reference

LoadVars.contentType Property Flash 6

the MIME type used for send( ) and sendAndLoad( ) operations *read/write*

loadVarsObject.contentType

Description

The `contentType` property is the MIME content type that is sent to a server when *LoadVars.send( )* or *LoadVars.sendAndLoad( )* is invoked. For most practical purposes, its default value (`application/x-www-form-urlencoded`) need never be changed. It is writable to allow MIME type flexibility in unique circumstances.

The `LoadVars.prototype.contentType` property can be used to change the MIME content type for all *LoadVars* instances that do not set it themselves. For example:

```
LoadVars.prototype.contentType = "yourContentTypeGoesHere";
```

Example

```
dataSender = new LoadVars();
trace(dataSender.contentType);  // Displays: application/x-www-form-urlencod
```

See Also

LoadVars.send( ) , *LoadVars.sendAndLoad( )*

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Chapter 18. ActionScript Language Reference

LoadVars.decode( ) Method Flash 6 (undocumented)

Convert a URL-encoded string of variables to properties

loadVarsObject.decode(*varString*)

Arguments

varString

A string of URL-encoded variables. For information on URL encoding, see the Description heading under the *LoadVars* class entry.

Description

The *decode( )* method converts the URL-encoded variables in *varString* to properties of the specified *loadVarsObject*. It offers a handy means of converting URL-encoded variables to object properties, as required by *LoadVars.onData( )*. Although undocumented, you can expect it to remain supported, because it is used internally by Flash to receive loaded variables.

Note that the property values created by *decode( )* are always strings.

Example

```
// Create a LoadVars instance
varsObj = new LoadVars();

// Convert a string to the object properties x and y
varsObj.decode("x=10&y=20");

// Check the value of y
trace(varsObj.y);  // Displays: 20
```

See Also

LoadVars.load( ), LoadVars.onData( )

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Chapter 18. ActionScript Language Reference

LoadVars.getBytesLoaded( ) Method Flash 6

check what portion of loading variables has arrived, in bytes

```
loadVarsObject.getBytesLoaded()
```

Returns

The number of bytes transferred so far, or `undefined` if neither *load( )* nor *sendAndLoad( )* has ever been called on *`loadVarsObject`* .

Description

The *getBytesLoaded( )* method tells us what portion of loading variables has arrived in the Player. For a new *LoadVars* ob *getBytesLoaded( )* returns `undefined` . Immediately after *load( )* or *sendAndLoad( )* is called, *getBytesLoaded( )* retur data arrives, *getBytesLoaded( )* returns the number of bytes transferred so far, until all the variables have loaded. From the returns the total number of bytes last transferred, until the next call to *load( )* or *sendAndLoad( )* . To convert bytes to kilo (KB), divide by 1024.

Along with *getBytesTotal( )* , *getBytesLoaded( )* is most commonly used to display the progress of variable retrieval, as sl the following Example.

Example

Example 18-1 displays load progress for variables loaded from a text file, *vars.txt* , which contains two URL-encoded va `msg1` and `msg2` . There are two items of special interest in the code:

- We use *sendAndLoad( )* , instead of *load( )* , to prevent unwanted caching. See *LoadVars.load( )* for details.
- We use separate objects for sending and receiving. See *LoadVars.sendAndLoad( )* for details.

Example 18-1. A LoadVars preloader

```
// Create a text field to display load progress
this.createTextField("loadmsg_txt", 1, 200, 200, 100, 20);

// Create an object to send variables
varSender = new LoadVars();

// Add a dummy property to prevent the browser from using its
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

```
// cache to retrieve variables
varSender.cacheKiller = new Date().getTime();

// Create an object to receive variables
varReceiver = new LoadVars();

// Assign a callback function to execute when the variables finish loading
varReceiver.onLoad = function () {
  // Display the loaded variables in the Output window
  trace("msg1 is " + this.msg1);
  trace("msg2 is " + this.msg2);

  // Loading is done, so stop calling our load checker function
  clearInterval(varPreloader);

  // Call checkVarStatus() at the end to display the final bytes loaded
  checkVarStatus();
}

// Request variables using sendAndLoad() instead
// of load() to prevent the browser from using its cache.
varSender.sendAndLoad("http://www.moock.org/temp/vars2.txt", varReceiver, "G

// Create an interval to check the variable load status every 100 ms
varPreloader = setInterval(checkVarStatus,100);

// Function to report variable load status
function checkVarStatus () {
  // Check bytes loaded and convert to KB
  var kbLoaded = Math.floor(varReceiver.getBytesLoaded()/1024);
  // Check bytes total and convert to KB
  var kbTotal = Math.floor(varReceiver.getBytesTotal()/1024);
  // If the bytes total can't be determined...
  if (kbTotal =  = undefined) {
    // ...use question marks instead of a total
    kbTotal = "???";
  }

  // Display the current load status in our text field
  loadmsg_txt.text = kbLoaded + "/" + kbTotal;
}
```

Bugs

Macromedia's documentation for Flash MX claims that *getBytesLoaded( )* returns `undefined` when no load operation progress. In Flash Player 6, it actually returns the total bytes transferred in the most recent load, even after that load has finished.

See Also

LoadVars.getBytesTotal( ) , *LoadVars.load( )* , *LoadVars.sendAndLoad( )* , *setInterval( )* , the *TextField* class

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Chapter 18. ActionScript Language Reference

LoadVars.getBytesTotal( ) Method Flash 6

check the expected total byte size of loading variables

```
loadVarsObject.getBytesTotal()
```

Returns

The total number of bytes being transferred as a result of a *load( )* or *sendAndLoad( )* call, or `undefined` before the number of bytes is determined.

Description

The *getBytesTotal( )* method tells us the expected total byte size of the current variable transfer. For a new *LoadVars* object, *getBytesTotal( )* returns `undefined`. Immediately after *load( )* or *sendAndLoad( )* are called, *getBytesTotal( )* continues to return `undefined` until the size of the variables being loaded is determined (normally within a second). It then returns the total byte size of the loading variables until the next call to *load( )* or *sendAndLoad( )*. If the total byte size cannot be determined (for example, due to a missing Content-Length in an HTTP response header), *getBytesTotal( )* continues to return `undefined`. To convert bytes to kilobytes (KB), divide by 1024.

For an example that uses *getBytesTotal( )* to create a variable preloader, see Example 18-1 under *LoadVars.getBytesLoaded( )*.

Usage

Be careful not to divide by *getBytesTotal( )* until it returns some positive integer.

See Also

LoadVars.getBytesLoaded( ), LoadVars.load( ), LoadVars.sendAndLoad( )

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

LoadVars.load( ) Method Flash 6

import URL-encoded variables from a text file or server-side application

```
loadVarsObject.load(url)
```

Arguments

url

A string specifying the path to a variable source—either a server-side script that returns variables or a text file cont variables.

Returns

A Boolean indicating whether the method executed successfully. Note that the return value is less useful than it appears. T *)* method returns `false` only if it is invoked with no arguments. Because loading is asynchronous, *load( )* 's return value only whether the load was initiated successfully; it will return `true` even if the load operation eventually fails, in which c *onLoad( )* is called with a `false` status.

Description

The *load( )* method initiates the download of URL-encoded variables from `url` , which can be a local or remote text file server-side application. Download progress can be monitored with the *getBytesLoaded( )* and *getBytesTotal( )* methods. V variables have finished downloading, they are parsed and converted into properties of `loadVarsObject` , and `loadVarsObject` *.onLoad( )* is invoked. All properties are typed as strings. To intercept the loaded data before it is converted into properties, use *LoadVars.onData( )* . Note that no variables are accessible as `loadVarsObject` proper before *onLoad( )* executes; all variables must be downloaded and parsed before any are available as properties. Likewise, variables must be downloaded before they are available to ActionScript in the *onData( )* event handler.

When `url` is a server-side application or script, the MIME type of the transmitted data must be set as `"applicatior www-form-urlencoded"` . Here's a typical MIME-setting statement from a Perl script:

```
print "Content-type: application/x-www-form-urlencoded\n\n";
```

For more information on URL encoding in Flash, see the Description heading of the *LoadVars* class.

For security reasons, *LoadVars.load( )* works only with hosts in the domain from which the movie was downloaded. For n information, see `System.security` .

Usage

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

In practice, the *load( )* and *sendAndLoad( )* methods may not trigger a download if the variables at `url` are cached by th browser hosting the Flash movie. To force the browser to always download variables (and bypass its cache) we can use a string with a dummy variable that contains a unique value. By convention, the current time in milliseconds is used, as it is to be sufficiently unique. For example:

```
theVars = new LoadVars();
theVars.load("http://www.site.com/vars.txt?cacheKiller=" + new Date().getTim
```

The same effect can be achieved using the *sendAndLoad( )* method to invoke an HTTP GET request with a dummy varial (`cacheKiller`):

```
varSender = new LoadVars();
varReceiver = new LoadVars();
varSender.cacheKiller = new Date().getTime();
varSender.sendAndLoad("http://www.yourserver.com/vars.txt", varReceiver, "GE
```

In all cases, data should be verified inside the *onLoad( )* handler before being used, because cached data can be corrupt or incomplete. See *LoadVars.onLoad( )* for an example.

Example

The following code uses a *LoadVars* object to retrieve the variable in the text file *vars.txt* , which looks like this:

```
msg1=hello+world
```

Two variations of *load( )* are shown, one with a unique cache-prevention variable:

```
// Make a text field in which to display the loaded variable
this.createTextField("msg1_txt", 1, 200, 200, 100, 20);

// Create the LoadVars instance
externalVars = new LoadVars();

// Assign an onLoad() callback to tell us that the variable has arrived
externalVars.onLoad = function () {
  // When the variable arrives, display it in our text field
  msg1_txt.text = this.msg1;
}

// Now load the text file
externalVars.load("http://www.site.com/vars.txt");

// Or, to bypass browser cache, append a unique dummy variable to the query
externalVars.load("http://www.site.com/vars.txt?cacheKiller="
                  + new Date().getTime());
```

See Also

loadVariables( ) , *LoadVars.getBytesLoaded( )* , *LoadVars.onData( )* , *LoadVars.sendAndLoad( )*

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Chapter 18. ActionScript Language Reference

LoadVars.loaded Property Flash 6

status of a load( ) or sendAndLoad( ) operation *read/write*

loadVarsObject.loaded

Description

The `loaded` property provides a quick means of determining whether variables have finished loading. For a new *LoadV* object, `loaded` is set to `undefined` . After the first call to *load( )* or *sendAndLoad( )* , `loaded` is set to `false` . If download never completes, due to an error or poor connection, it remains `false` . If download does complete, `loaded` to `true` by *loadVarsObject .onData( )* and remains true until the next call to *load( )* or *sendAndLoad( )* .

In most cases, *LoadVars.onLoad( )* provides the best way to respond when variable loading completes. However, the `loa` property is handy when a separate segment of code needs immediate access to the load status of a *LoadVars* object, as des in the following Example.

Example

Consider a game that loads high scores and offers a button to email scores to a friend. The button should work only if the s are loaded, as follows:

```
// highScoreVars is assumed to be a global variable containing a LoadVars ob
emailButton.onRelease = function () {
  if (highScoreVars.loaded =  = true) {
    // Proceed with email code...
  } else {
    // Variables aren't loaded. Tell user to wait...
  }
}
```

See Also

LoadVars.onLoad( )

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Chapter 18. ActionScript Language Reference

LoadVars.onData( ) Event Handler Flash 6 (undocumented)

handler executed when external variable data finishes loading, but before it is parsed

```
loadVarsObject.onData(src);
```

Arguments

src

A string containing raw URL-encoded variables.

Description

The *onData( )* handler executes automatically when raw variables requested by *load( )* or *sendAndLoad( )* have finished downloading but have not yet been parsed.

The *onData( )* handler can be assigned a custom callback function to intercept raw variable source before ActionScript has a chance to convert the URL-encoded variables to `loadVarObject` properties and before *onLoad( )* is called. This unmediated access to data loaded from the server can be useful for debugging. However, in most other cases a custom callback for *onData( )* is not required.

By default (unless you define a custom callback function), *onData( )* has the following behavior:

- If the raw source received is `undefined`, it calls `loadVarsObject`.*onLoad( )* with the `success` parameter set to `false`.
- Otherwise, it uses `loadVarsObject`.*decode( )* to convert the variables in `src` to properties of `loadVarsObject`, then sets `loadVarsObject.loaded` to `true`, and calls `loadVarsObject`.*onLoad( )* with the `success` parameter set to true.

When implementing *onData( )*, be sure to convert the loaded variables to properties via *decode( )*, as shown in the following Example.

Example

The following example shows how to display raw loaded-variable source while preventing it from being parsed by ActionScript. The *onLoad( )* method is not called, and the `loaded` property is not set:

```
loadedVars = new LoadVars();
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

```
loadedVars.onData = function (src) {
  trace("Variables received: " + src);
};
loadedVars.load("http://www.thesite.com/cgi-bin/getData.pl");
```

Here we display raw loaded-variable source but call *onLoad( )* as it normally would be. Note that we take care to recreate all the default *onData( )* behavior, including converting the loaded variables to properties with *decode( )* and setting the `loaded` property to `true`:

```
loadedVars = new LoadVars();
loadedVars.onData = function (src) {
  trace("Variables received: " + src);
  if (src =  = undefined) {
    this.onLoad(false);
  } else {
    this.decode(src);
    this.loaded = true;
    this.onLoad(true);
  }
};

loadedVars.onLoad = function (success) {
  trace("Vars load successfully? " + success);
}

loadedVars.load("http://www.thesite.com/cgi-bin/getData.pl");
```

See Also

LoadVars.decode( ), LoadVars.onLoad( )

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

LoadVars.onLoad( ) Event Handler Flash 6

handler executed after external variables have been converted into object properties

loadVarsObject.onLoad(*success*)

Arguments

success

A Boolean value indicating whether loading was successful (`true`) or unsuccessful (`false`).

Description

The *onLoad( )* handler executes automatically whenever external variables have finished loading and have been converted (parsed) into *`loadVarsObject`* properties. We rely on *onLoad( )* to tell us when it's safe to process loaded variables; if *onLoad( )* is triggered, we know that we can access our loaded content. By default, the *onLoad( )* handler of a *LoadVars* object is an empty function. To use *onLoad( )* , we must assign it a callback function. For example:

```
// Create the LoadVars instance
varReceiver = new LoadVars();

// Assign the callback function to onLoad()
varReceiver.onLoad = handleLoad;

// Define the callback function
function handleLoad (success) {
  // Process variables as desired here...
}
```

This syntax, which uses a function literal, is also common:

```
varReceiver.onLoad = function (success) {
  // Process variables as desired here...
}
```

From inside our callback function, we access *`loadVarsObject`* 's properties via the `this` keyword. For example:

```
function handleLoad (success) {
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

```
  // Display the value of the loaded variable score in the Output window
  trace(this.score);
}
```

The *success* parameter tells us whether any variables were actually loaded. If the data loaded from the server was empty, *success* is `false` ; otherwise, it's `true` . We can use *success* to provide basic verification that something useful was actually loaded, as follows:

```
function handleLoad (success) {
  if (success) {
    // Display the value of the loaded variable score in the Output window
    trace(this.score);
  } else {
    trace("No variables were found");
  }
}
```

Note, however, that a *success* of `true` does not guarantee that the variables were properly formatted or useful, it merely indicates that the HTTP response's content was not empty. Occasionally, variable data is not received properly even when *success* is `true` . For example, a server-side script may have sent the wrong data, or stale data may have been read out of the browser cache. It is good practice to test explicitly for the variables we expect to receive before using them. For example, here we test for the variable `msg` before using it:

```
function handleLoad (success) {
  if (success) {
    // Check for msg
    if (this.msg != undefined) {
      // It's safe to proceed...
    } else {
      // msg didn't arrive...abort, or try loading the variables again...
    }
  } else {
    trace("No variables were found");
  }
}
```

The *onLoad( )* handler alleviates the need to write custom polling code that waits for data to arrive after invoking *load( )* or *sendAndLoad( )* . For a complete example showing *onLoad( )* and variable preloading techniques, see *LoadVars.getBytesLoaded( )* and *LoadVars.sendAndLoad( )* .

See Also

LoadVars.getBytesLoaded( ) , LoadVars.load( ) , LoadVars.onData( ) , LoadVars.sendAndLoad( )

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Chapter 18. ActionScript Language Reference

LoadVars.send( ) Method — Flash 6

transfer URL-encoded variables to a server-side application or script

```
loadVarsObject.send(url, target)
loadVarsObject.send(url, target, method)
```

Arguments

url

A string specifying the absolute or relative location of the external script or application that receives the variables.

target

A string specifying the name of the browser window or frame into which to load results. Can be a custom name or the four presets: `"_blank"` , `"_parent"` , `"_self"` , or `"_top"` .

method

An optional string specifying the HTTP method by which to send the properties of *`loadVarsObject`* to an ex script—either `"GET"` or `"POST"` . If omitted, it defaults to `"POST"` . The Standalone Flash Player for Windows always uses the `"GET"` method, regardless of the *`method`* specified. See Appendix F for details on support for and POST.

Returns

A Boolean indicating whether the method executed successfully. Note that the return value is less useful than it appears. V passed *any* value for *`url`* (including `undefined` !), *send( )* returns `true` . When invoked with no arguments, *send( )* `false` .

Description

The *send( )* method converts all properties of *`loadVarsObject`* to a string of URL-encoded variables, which it then s to *`url`* in an HTTP request, using the method specified by *`method`* . If the script or application at *`url`* outputs a resu displayed in the browser frame or window specified by *`target`* . If the movie is running in the Standalone Player, the s default browser is launched. See the global *getURL( )* function for a detailed explanation of the *`target`* and *`method`* parameters. For an explanation of URL encoding, see the Description heading of the *LoadVars* class. The URL-encoded s sent to *`url`* can be viewed locally via the *LoadVars.toString( )* method.

Note that the *getURL( )* function also can pass variables to a remote script, but it is less convenient than *LoadVars.send( )* because it requires the use of a movie clip to hold the variables.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Bugs

Macromedia's documentation states that when the `target` parameter is not specified, the script at `url` is executed, bu results are discarded. However, in Flash Player 6, tests show that when `target` is not specified, the *send( )* the method execute the script at `url` .

Example

The following code uses a *LoadVars* object to send the properties `name` and `age` as URL-encoded variables to the script *userFind.pl* :

```
// Create the LoadVars object
varSender = new LoadVars();

// Assign properties to the varSender object
// These are the variables sent to the server
varSender.name = "Bruce";
varSender.age = "13";

// Send variables to server, and catch reply in a new browser window
varSender.send("http://www.someserver.com/cgi-bin/userFind.pl", "_blank", "G
```

See Also

getURL( ) , LoadVars.onLoad( ) , LoadVars.sendAndLoad( ) , LoadVars.toString( )

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

LoadVars.sendAndLoad( ) Method Flash 6

transfer variables to a server-side application and receive variables in return

```
loadVarsObject.sendAndLoad(url, targetObject)
loadVarsObject.sendAndLoad(url, targetObject, method)
```

Arguments

url

A string specifying the absolute or relative location of the external script or application that receives the Flash vari
targetObject

A reference to the *LoadVars* object that will receive the server's response.
method

An optional string specifying the HTTP method by which to send the properties of `loadVarsObject` to an ex
script—either `"GET"` or `"POST"` . If omitted, it defaults to `"POST"` . Unlike the *send( )* method, *sendAndLoad(*
`"POST"` successfully in all versions of the Standalone Player. See Appendix F for details on support for GET and

Returns

A Boolean indicating whether the method executed successfully. Note that because the *sendAndLoad( )* operation is asyn
the return value indicates only whether the operation commenced successfully; it may still eventually fail, in which case *o*
will be invoked with a `success` value of `false` . The *sendAndLoad( )* method returns `false` if it is passed the inco
number of arguments (must be two or three), if the `targetObject` parameter is not an object, or if the request is prol
to security restrictions.

Description

The *sendAndLoad( )* method converts all properties of `loadVarsObject` (excluding the `contentType` and `loa`
properties) to a string of URL-encoded variables, which it then sends to `url` in an HTTP request, using the method spec
`method` . The script or application at `url` is expected to return a string of URL-encoded variables, which are downlo
the Player and converted to `targetObject` properties, as described under *LoadVars.load( )* . The `targetObject`
onLoad( ) handler executes when the properties become available. All properties are typed as strings.

Though a single *LoadVars* object can be used to both send and receive variables, it's best to separate the two operations in
LoadVars objects, as shown in the following Example. This keeps outgoing and incoming data confined to its own object
prevents unwanted transmission of the receiving object's properties, such as the `loadVarsObject.onLoad` handle

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

For security reasons, *LoadVars.sendAndLoad( )* works only with hosts in the domain from which the movie was downloa more information, see `System.security` .

Usage

In practice, the *sendAndload( )* and *load( )* methods may not trigger a download if the variables at `url` are cached by the hosting the Flash movie. See *LoadVars.load( )* for techniques that force the browser to bypass its cache.

Example

The following code uses a *LoadVars* object to send variables to the script *lookup.pl* . The `varReceiver` object catches response sent by *lookup.pl* :

```
// Create the LoadVars objects. One to send...
varSender = new LoadVars();
// ...and one to receive.
varReceiver = new LoadVars();

// Assign properties to the sender, which will be the variables sent to the
varSender.name = "Bruce";
varSender.age = "13";

// Assign a callback function to the varReceiver's onLoad property.
// The handler will process the variables when they arrive.
varReceiver.onLoad = function () {
  // We can access the loaded variables as properties of this, as in this.ag
  // We use a for-in loop to list the properties of the varReceiver object.
  // Note that the loaded and contentType properties are not enumerated.
  for (p in this) {
    // We don't want to list the onLoad property, so check if the
    // current property is a string. If it is...
    if (typeof this[p] =  = "string") {
      // ...show its value in the Output window. All loaded
      // variables are received as strings.
      trace("The variable " + p + " has the value " + this[p]);
    }
  }
}

// Before sending our variables, we'll display them
// in the Output window. (Good for debugging)
trace("Sending " + varSender.toString());

// Now send the variables to the server, and wait for a reply
varSender.sendAndLoad("http://www.yourserver.com/cgi-bin/lookup.pl",
                      varReceiver, "GET");
```

See Also

LoadVars.load( ) , *LoadVars.onLoad( )* , *LoadVars.send( )* , *LoadVars.toString( )*

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Chapter 18. ActionScript Language Reference

LoadVars.toString( ) Method Flash 6

retrieve the same URL-encoded string of variables that would be sent by send( ) or sendAndLoad( )

loadVarsObject.toString()

Returns

A string of URL-encoded variables.

Description

The *toString( )* method returns the enumerable properties of *loadVarsObject* as a string of URL-encoded variables enumerable properties are those exposed to a *for-in* loop). It returns the same data that would be sent to a server script in a *)* or *sendAndLoad( )* call, which is handy for debugging. Properties are added to the returned string in the reverse order of creation (the most recently created property is first in the string, followed by the second most recently created, and so on). property's value is converted to a string before being URL encoded.

Example

```
loadedVars = new LoadVars();
loadedVars.day = 22;
loadedVars.month = "Feb.";
loadedVars.year = 2002;

// Displays: year=2002&month=Feb%2E&day=22
// Notice that the '.' after "Feb" is converted to the hex value %2E.
// See the Description heading of the LoadVars class for details on URL enco
trace(loadedVars.toString());
```

See Also

LoadVars.onLoad( ) , LoadVars.send( ) , LoadVars.sendAndLoad( )

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Chapter 18. ActionScript Language Reference

LocalConnection Class Flash 6

transmit data directly between movies running on the same system

Constructor

`new LocalConnection()`

Methods

close( )

Stop a connection from receiving messages.

connect( )

Open a named connection.

domain( )

Returns a string indicating the movie's subdomain.

send( )

Invoke a method on a remote *LocalConnection* object.

Event Handlers

allowDomain( )

Callback to permit or deny a cross-domain connection attempt.

onStatus( )

Callback indicating initial status of a *send( )* invocation.

Description

The *LocalConnection* class lets movies running in different Flash Players communicate with each other. The Players must be running on the same computer, but they can be embedded in any application. For example, movies running in the Netscape Flash 6 plugin, the Internet Explorer ActiveX Flash Player, and the Standalone Flash Player can all communicate with each other as long as they are running on the same machine. To transfer information between two Flash Players running on different machines use *XML*, *LoadVars*, *XMLSocket*, or the Macromedia Flash Communication Server MX (Comm Server).

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

All *LocalConnection* applications involve at least two *LocalConnection* objects, one in the *receiver* (the movie that receives messages) and one in the *sender* (the movie that sends messages). Although, in practice, both movies may act as both senders and receivers, let's start with simple one-way communication for illustration purposes.

The first line of code in the receiver is:

```
receiveConn = new LocalConnection();
```

And the first line in the sender is:

```
sendConn = new LocalConnection();
```

Once the receiver and sender have each created a *LocalConnection* object, they must agree in advance on a name for the connection. On the receiver's end, the connection name is specified when the connection is opened. However, the connection should not be opened until the receiver has defined methods that the sender will invoke remotely. These methods form the basis of all intermovie communication. For example, here we define a method on our `receiveConn` object that displays a supplied argument in a text field:

```
// Create the text field
this.createTextField("received_txt", 1, 50, 100, 400, 100);
received_txt.border = true;

// Define the method
receiveConn.displayMsg = function (msg) {
  received_txt.text = msg;
}
```

With the *receiveConn.displayMsg( )* method defined, we now can safely open the receiver's connection. On the receiver side, we use the *connect( )* method to start listening for messages. The connection name is supplied as an argument to the *connect( )* method; in the following line, it is "channel1":

```
receiveConn.connect("channel1");
```

Our receiver movie is now open for business. All that's left to do is send a request from the sender to invoke *displayMsg( )* on the receiver movie. The sender uses the *send( )* method to perform remote method invocations. Arguments to *send( )* specify the connection name ("channel1"), the method to invoke ("displayMsg"), and the method argument(s) ("hello world"), as follows:

```
sendConn.send("channel1", "displayMsg", "hello world");
```

Here is the complete code required to establish basic *LocalConnection* communication:

```
// CODE IN THE RECEIVING MOVIE
receiveConn = new LocalConnection();
receiveConn.methodName = function (param1, param2) {
  // statements
}
receiveConn.connect("connectionName");

// CODE IN THE SENDING MOVIE
sendConn = new LocalConnection();
sendConn.send("connectionName", "methodName", "value1", "value2");
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

For security reasons, when the sender and receiver are posted at different domains, they cannot, by default, communicate over a *LocalConnection.* See *LocalConnection.allowDomain( )* for details on allowing movies to communicate across domains. All movies running on the local filesystem are considered in the same domain ("localhost"), and are therefore not subject to security restrictions.

Communication over *LocalConnection* is many-to-one, from the sender's perspective. Any number of senders can send messages to a specific named connection, but only one receiver can listen on that connection. If a different receiver tries to open a named connection that is already open, the latter connection attempt fails, and the *connect( )* method returns `false`.

A *LocalConnection* object can act as both a sender and a receiver, allowing two objects to engage in bidirectional communication. For sample code, see the Example heading under *LocalConnection.onStatus( ).*

Prior to Flash 6, local intermovie communication could be achieved only with *fscommand( )* and JavaScript, and only in specific browser/platform combinations. See *fscommand( )* for details.

See Also

fscommand( )

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Chapter 18. ActionScript Language Reference

LocalConnection.allowDomain( ) Event Handler Flash 6

callback to permit or deny a cross-domain connection attempt

receiveConn.allowDomain(*domain*)

Arguments

domain

A string representing the sender's subdomain.

Returns

`true` if the connection is allowed, otherwise `false` .

Description

The *allowDomain( )* event is triggered on a receiving *LocalConnection* object whenever a message is received. The *allowDomain( )* callback is expected to permit or deny the connection attempt by returning a value of `true` (permitted) o `false` (denied). When the sender and receiver are loaded from the same domain, including the same localhost, Flash automatically permits all connections, and *allowDomain( )* should not be used. However, when the sender and receiver ar loaded from separate domains, a connection is not allowed unless *allowDomain( )* is implemented and returns `true` . Wh implemented, *allowDomain( )* normally compares the value of its *domain* parameter with a list of trusted hosts. Note th *LocalConnection.allowDomain( )* is a callback handler that returns whether to allow a connection. Unlike *Security.settings.allowDomain( )* , it is not a method to be configured once with the names of allowed subdomains.

The following receiver movie code creates a *LocalConnection* object that opens a connection named "channel1":

```
// Create a text field to display messages
this.createTextField("output_txt", 1, 50, 100, 400, 100);
output_txt.border = true;

// Create the connection object
receiveConn = new LocalConnection();
// Assign a method to be invoked remotely
receiveConn.displayMsg = function (msg) {
  output_txt.text += msg + "\n";
}

// Attempt to open a connection named "channel1". Store the result in a vari
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

```
connectSuccess = receiveConn.connect("channel1");
// Display whether the connection succeeded
output_txt.text = "Receiver connection succeeded? " + connectSuccess + "\n";
```

Now suppose we post this receiver movie at http://www.graydient.com, and we want to allow messages from *moock.org* c (not even from *graydient.com* !). Here is the callback we assign to *allowDomain( )* :

```
// Handle incoming messages
receiveConn.allowDomain = function (domain) {
  // If this handler is running, a new message arrived.
  // Show where the message is from.
  output_txt.text += "New message from: " + domain + "\n";

  // Now the key decision: if the message is from moock.org...
  if (domain =  = "moock.org") {
    // ...permit it
    return true;
  }  else {
    // ...otherwise deny it
    return false;
  }
}
```

If we want to allow messages from *moock.org* and the current receiving movie's subdomain (in this case *graydient.com*), use the *domain( )* method, which returns a string indicating the movie's subdomain:

```
// Handle incoming messages
receiveConn.allowDomain = function (domain) {
  // Show where the message is from
  output_txt.text += "New message from: " + domain + "\n";

  // If the message is from moock.org or the receiver's domain...
  if (domain =  = "moock.org" || domain =  = this.domain()) {
    // ...permit it
    return true;
  }  else {
    // ...otherwise deny it
    return false;
  }
}
```

Or, more succinctly:

```
receiveConn.allowDomain = function (domain) {
  return domain =  = "moock.org" || domain =  = this.domain();
}
```

Our receiver work is done, but there's still a minor issue to contend with—the sender movie must identify the connection r correctly when sending messages. When it opened the connection, `receiveConn` set the connection name as "channel in order to prevent connection name conflicts across domains, Flash automatically prefixes the connection name with the r subdomain and a colon. For example, when our receiver movie is posted at *graydient.com* , the actual connection name is "channel1", but "graydient.com:channel1". Hence, when specifying the connection name in the sender, we must use:

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

```
sendConn = new LocalConnection();
// Add domain name of receiver to connection name
sendConn.send("graydient.com:channel1", "displayMsg", "hello world");
```

If we know our receiver will receive messages only while posted at *graydient.com* , we're done. However, if we want to re connections no matter where our receiver is posted, the plot gets a little thicker. For example, suppose we want to run a re both on the local filesystem (domain name "localhost") and on *graydient.com* . When we invoke the *connect( )* method frc movie residing on *graydient.com* :

```
receiveConn.connect("channel1");
```

the final connection name becomes "graydient.com:channel1". But when we invoke the *connect( )* method from a movie residing on the local filesystem, the final connection name becomes "localhost:channel1". Obviously, the sender cannot sp both connection names at once. The solution is to instruct Flash not to prefix the connection name with the domain name, adding a leading underscore (_). For example:

```
receiveConn.connect("_channel1");
```

With no prefix domain, the sender can now address the connection directly as follows:

```
sendConn.send("_channel1", "displayMsg", "hello world");
```

Note, however, that names starting with an underscore will conflict across domains. For example, if a receiver at *oreilly.c* opens a connection named "_bookComm", and then *moock.org* opens a connection by the same name, the connection at *moock.org* will fail. Hence, use unique names when a leading underscore is specified (for example: "_org.moock.channel

See Also

LocalConnection.connect( ) , *LocalConnection.domain( )* , *LocalConnection.send( )* , *Security.settings.allowDomain( )*

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Chapter 18. ActionScript Language Reference

LocalConnection.close( ) Method Flash 6

stop a connection from receiving messages

```
receiveConn.close()
```

Returns

A Boolean—`true` if an open connection was closed, `false` if no open connection was found for *`receiveConn`*.

Description

The *close( )* method is used only by a receiving *LocalConnection* object to close an open connection, thus preventing future senders from invoking its methods. For example, the following code opens a connection named "channel2", making *displayMsg( )* eligible for remote execution:

```
receiveConn = new LocalConnection();
receiveConn.displayMsg = function (msg) {
  output_txt.text += msg + "\n";
}
receiveConn.connect("channel2");
```

Later, we can close the "channel2" connection by invoking *close( )*:

```
receiveConn.close();
```

Closing the connection prevents all senders from invoking *displayMsg( )*. A *LocalConnection* object can maintain only one open connection at a time. Hence, *close( )* does not take the name of a connection to close as a parameter. Note that *close( )* applies only to receivers; a *LocalConnection* object that only sends messages need not open or close a connection.

See Also

LocalConnection.connect( )

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Chapter 18. ActionScript Language Reference

LocalConnection.connect( ) Method Flash 6

open a named connection

```
receiveConn.connect(connectionName)
```

Arguments

connectionName

A string name for the connection.

Returns

A Boolean—`true` if the connection opened successfully, `false` if the operation failed (for example, if a connection named *`connectionName`* already exists in any Player or if *`receiveConn`* already has an open connection).

Description

The *connect( )* method is used by a receiving *LocalConnection* object to open a connection, thus enabling future senders to invoke its methods. All methods should be defined before *connect( )* is invoked. For example, the following code creates a *LocalConnection* object, then defines its methods, then opens a connection named "channel3" (making *displayMsg( )* eligible for remote execution):

```
receiveConn = new LocalConnection();
receiveConn.displayMsg = function (msg) {
  output_txt.text += msg + "\n";
}
receiveConn.connect("channel3");
```

By checking the return value of *connect( )*, we can respond gracefully to connection attempt failures. For example:

```
connectSuccess = receiveConn.connect("channel3");
if (connectSuccess) {
  // ...proceed with application
} else {
  // ...display failure screen
}
```

When *`connectionName`* does not start with an underscore (_), the name specified is automatically

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

prefixed with `receiveConn`'s subdomain and a colon (e.g., "channel3" becomes "oreilly.com:channel3" if the movie is posted at *www.oreilly.com*). When `connectionName` starts with an underscore, no prefix is applied.

Senders invoking remote methods on `receiveConn` must match the final connection name (automatic prefix + `connectionName`) when invoking *send( )*. See *LocalConnection.allowDomain( )* and *LocalConnection.send( )* for a discussion of connection name matching.

A *LocalConnection* object can maintain only one open connection at a time and cannot open a connection with a connection name that is already in use. To close an open connection, use *LocalConnection.close( )*.

See Also

LocalConnection.allowDomain( ), *LocalConnection.close( )*, *LocalConnection.send( )*

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Chapter 18. ActionScript Language Reference

LocalConnection.domain( ) Method Flash 6

returns a string indicating the movie's subdomain

```
localConn.domain()
```

Returns

The movie's subdomain, as a string.

Description

The *domain( )* method simply returns the so-called *subdomain* of the current movie, as a string. For one- and two-component domain names, such as *apollo* and *oreilly.com*, the subdomain is identical to the complete domain. For domain names with three or more components, such as *www.moock.org* or *events.news.thedailybugle.com*, the subdomain is defined as everything after the first dot, in this case "moock.org" and "news.thedailybugle.com". For movies running on the local filesystem, *domain( )* returns "localhost". The subdomain is used for a variety of security checks in the Flash Player, including *LoadVars* and *XML* network communication.

The *domain( )* method is used by *LocalConnection* objects communicating across domains. It lets us specify the subdomain of a sender or receiver movie without knowing its name beforehand. For example, the following code allows messages from a sender at either *oreilly.com* or the same domain as the receiver (whatever that domain happens to be):

```
receiveConn.allowDomain = function (domain) {
  return domain =  = "oreilly.com" || domain =  = this.domain();
}
```

See Also

LocalConnection.allowDomain( ), LocalConnection.send( ), `MovieClip._url`, *System.security.allowDomain( )*

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Chapter 18. ActionScript Language Reference

LocalConnection.onStatus( ) Event Handler Flash 6

callback indicating initial status of a send( ) invocation

```
sendConn.onStatus(infoObject)
```

Arguments

infoObject

An object indicating the status of the most recent *send( )* operation.

Description

The *onStatus( )* event is triggered whenever *LocalConnection.send( )* is invoked by `sendConn` . It tells us whether the *se* operation found a valid connection that matched the specified `connectionName` . The *onStatus( )* callback receives a argument, `infoObject` , whose `level` property will have the value "error" if the specified `connectionName` w found, or "status" if it was found. This handler is used for debugging and error-handling code. For example:

```
sendConn.onStatus = function (infoObject) {
  if (infoObject.level =  = "status") {
    trace("Connection found.");
  } else if (infoObject.level =  = "error") {
    trace("No connection found.");
    // ...handle the failed connection attempt...
  }
}
```

Even if `infoObject .level` is "status" (a connection is found), the connection may not be accepted by the receiver, receiver may not define the specified method. For information on allowing cross-domain connections, see *LocalConnection.allowDomain( )* . To verify that a method was successfully invoked, the receiver must call a reply metho sender, as shown in the following Example.

Example

The following code shows a sender object that invokes *displayMsg( )* on a receiver object. The receiver object responds b *methodReturn( )* on the sender. If no connection is found, the sender displays a debugging message. Notice how each *LocalConnection* object acts as both a sender and a receiver, but we'll refer to the primary sender as "the sender" and the p receiver as "the receiver" for convenience. In the following example, the remote invocation is transacted over "channelA" opened by the receiver and accessed by the sender), and the response is transmitted over "channelB" (which is opened by and accessed by the receiver). This relies on both the sender and receiver agreeing on the channel names ahead of time. Furthermore, it assumes that both the sending and receiving movies are hosted on the same subdomain. Refer to the Discu

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

under *LocalConnection.send( )* for dealing with senders and receivers hosted on different domains.

Here is the code in the sender:

```
// Create a text field to display messages
this.createTextField("sendOutput_txt", 1, 50, 100, 400, 100);
sendOutput_txt.border = true;

// Create the sender LocalConnection object
sendConn = new LocalConnection();
// Create a method to catch receiver method return values
sendConn.methodReturn = function (result) {
  sendOutput_txt.text += result + "\n";
}

// Attempt to open a connection named "channelB" to be used for the response
connectSuccess = sendConn.connect("channelB");

// Display whether the connection succeeded
sendOutput_txt.text = "Sender connection succeeded: " + connectSuccess + "\n

// Invoke displayMsg() remotely using "channelA",
// which hopefully the receiver opened
sendConn.send("channelA", "displayMsg", "hello world");

// Check whether the connection is available
sendConn.onStatus = function (infoObject) {
  if (infoObject.level =  = "status") {
    sendOutput_txt.text += "Connection found.\n";
  } else if (infoObject.level =  = "error") {
    sendOutput_txt.text += "No connection found.\n";
  }
}
```

And here is the code in the receiver:

```
// Create a text field to display messages
this.createTextField("receiveOutput_txt", 1, 50, 100, 400, 100);
receiveOutput_txt.border = true;

// Create the LocalConnection object
receiveConn = new LocalConnection();

// Assign a method to be invoked remotely
receiveConn.displayMsg = function (msg) {
  receiveOutput_txt.text += msg + "\n";
  // Let the sender know the method was called. Response is sent on "channel
  this.send("channelB", "methodReturn", "displayMsg() called");
}

// Attempt to open a connection named "channelA" to be used to receive messa
connectSuccess = receiveConn.connect("channelA");
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

```
// Display whether the connection succeeded
receiveOutput_txt.text = "Receiver connection succeeded: " + connectSuccess
```

See Also

LocalConnection.allowDomain( ) , LocalConnection.connect( ) , LocalConnection.domain( ) , LocalConnection.send( )

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

LocalConnection.send( ) Method Flash 6

invoke a method on a remote LocalConnection object

```
localConn.send(connectionName, methodName, arg1, arg2,...argn)
```

Parameters

connectionName

The string name of the connection on which to invoke `methodName`, as set by the receiver's *connect( )* method.

methodName

The string name of the method to invoke on the receiver. Must not be one of the following reserved words: "send", "connect", "close", "domain", "onStatus", or "allowDomain".

arg1, ...argn

Optional arguments to pass to `methodName`.

Returns

A Boolean—`true` if the *send( )* structure is syntactically correct, otherwise `false`.

Description

The *send( )* method sends a request to invoke the `methodName` at the connection specified by `connectionName`. The *send( )* method can transfer a maximum of 40 KB of data. The optional arguments passed can be any datatype (Boolean, object, number, etc.). If an object is sent, its class is retained only if it is registered on both sides of the *LocalConnection* using *Object.registerClass( )*. See *Object.registerClass( )* for more details.

When the sender and receiver movies are both loaded from the same domain, the `connectionName` for *send( )* should simply match the `connectionName` specified by the receiver's *connect( )* method. When the sender and receiver movies are loaded from different domains, the final connection name generated by Flash for the sender must match the final name generated for the receiver.

For the sender, the final connection name is generated as follows:

- If the `connectionName` passed to *send( )* does not start with an underscore (_) and contains no colon (:), the sender's subdomain and a colon are prefixed to `connectionName`. For example, a

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

`connectionName` of "channel1" resolves to a final name of `"senderdomain.com:channel1"` (where `senderdomain.com` is the subdomain of the sending movie).

- If the `connectionName` passed to *send( )* contains a colon, the specified value is used verbatim. For example, a `connectionName` of "moock.org:channel1" remains unchanged.
- If the `connectionName` passed to *send( )* starts with an underscore, the specified value is used verbatim. For example, a `connectionName` of "_channel1" remains unchanged.

For the receiver, the final connection name is generated as follows:

- If the `connectionName` passed to *connect( )* does not start with an underscore and contains no colon, the receiver's subdomain and a colon are prefixed to `connectionName`. For example, a `connectionName` of "channel1" resolves to a final name of `"receiverdomain.com:channel1"` (where `receiverdomain.com` is the subdomain of the receiving movie).
- If the `connectionName` passed to *connect( )* starts with an underscore, the specified value is used verbatim. For example, a `connectionName` of "_channel1" remains unchanged.

On the receiver, the `connectionName` format `"moock.org:channel1"` cannot be used (the receiver is not allowed to specify its own subdomain). Table 18-10 shows the specified and final connection name values for both the sender and receiver, and indicates whether each combination would match when the movies are running on the same domain or on different domains. For a full discussion of connection name matching, including overriding the security restrictions across domains, see *LocalConnection.allowDomain( )*.

Table 18-10. Connection name matching

Sender specified	Sender final	Receiver specified	Receiver final	Same domain match?	Different domain match?
name	senddomain:name	name	recdomain:name	Yes	No (The automatic subdomain prefixes would not match.)
name	senddomain:name	_name	_name	No (No subdomain is used by the receiver.)	No
_name	_name	name	recdomain:name	No (No subdomain is used by the sender.)	No
_name	_name	_name	_name	Yes	Yes (But *allowDomain( )* must permit the connection.)

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

subdomain:name	subdomain:name	name	subdomain:name	Yes, if sender's specified subdomain matches receiver's actual subdomain	Yes (But *allowDomain( )* must permit the connection, and sender's specified subdomain must match receiver's actual subdomain.)
subdomain:name	subdomain:name	_name	_name	No (No subdomain is used by the receiver.)	No

See Also

LocalConnection.allowDomain( ), *LocalConnection.connect( )*, *LocalConnection.domain( )*

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Chapter 18. ActionScript Language Reference

Math Object — Flash 5; can be used when exporting Flash 4 movies

access to mathematical functions and constants

```
Math.propertyName
Math.methodName( )
```

Properties

E

The constant *e*, the base of natural logarithms, approximately 2.71828.

LN10

The natural logarithm of 10 ($\log_e 10$), approximately 2.30259.

LN2

The natural logarithm of 2 ($\log_e 2$), approximately 0.69315.

LOG10E

The base-10 logarithm of *e*, approximately 0.43429.

LOG2E

The base-2 logarithm of *e*, approximately 1.44270.

PI

The ratio of a circle's circumference to its diameter, approximately 3.14159.

SQRT1_2

The reciprocal of the square root of 2, approximately 0.70711.

SQRT2

The square root of 2, approximately 1.41421.

Methods

abs( )

Compute the absolute value of a number.

acos( )

Compute the arc cosine of a number.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

asin( )

Compute the arc sine of a number.

atan( )

Compute the arc tangent of a number.

atan2( )

Compute the angle of a point, relative to the x-axis.

ceil( )

Round a number up to the next integer.

cos( )

Compute the cosine of an angle.

exp( )

Raise *e* to a specified power.

floor( )

Return the closest integer less than or equal to the input.

log( )

Compute the natural logarithm of a number.

max( )

Determine the larger of two numbers.

min( )

Determine the smaller of two numbers.

pow( )

Raise a number to a specified power.

random( )

Retrieve a random floating-point number between 0 and 1.

round( )

Calculate the closest integer to a number.

sin( )

Compute the sine of an angle.

sqrt( )

Compute the square root of a number.

tan( )

Compute the tangent of an angle.

Description

The *Math* object provides access to built-in mathematical functions (accessed through methods) and constant values (accessed through properties). These functions and constants are used to perform potentially complex

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

calculations with relative ease.

Note that the properties and methods of the *Math* object can be used in movies exported in Flash 4 format, in which case Flash will approximate the calculations. The resulting values are reasonable approximations, but they are not necessarily identical to the native functions supported in Flash 5 and later. The Flash 4 values are sufficiently accurate for "close-enough" applications, such as graphics display, but are not accurate enough for critical financial or engineering calculations.

Note that the trigonometric functions require angles to be measured in radians, whereas Flash's *MovieClip*._rotation property is measured in degrees. There are 2 radians in a circle (1 radian is approximately 57.3 degrees). To convert from radians to degrees, use the formula:

```
degrees = (radians / Math.PI) * 180;
```

To convert from degrees to radians, use the formula:

```
radians = (degrees / 180) * Math.PI;
```

Flash automatically converts all *Math* object constants to literal numbers at compile-time.

See Also

Section 4.1

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Chapter 18. ActionScript Language Reference

Math.abs( ) Method — Flash 5; can be used when exporting Flash 4 movies

compute the absolute value of a number

```
Math.abs(x)
```

Arguments

x

Any integer or floating-point number.

Returns

The absolute value of *x* (a positive number of magnitude *x*).

Description

The *abs( )* method calculates the distance between *x* and 0 (i.e., the *absolute* value of *x*). It leaves zero and positive numbers unchanged and converts negative numbers into positive numbers of the same magnitude. It is useful for calculating the difference between two numbers without regard to which is larger. For example, it is useful when calculating the distance between two points, because distances are always positive.

Example

```
Math.abs(-5);  // Returns 5

// Calculate the difference between two numbers
function diff (num1, num2) {
  return Math.abs(num1-num2);
}

diff(-5, 5);  // Returns 10
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Chapter 18. ActionScript Language Reference

Math.acos( ) Method

Flash 5; can be used when exporting Flash 4 movies

compute the arc cosine of a number

```
Math.acos(x)
```

Arguments

x

A number between -1.0 and 1.0 (the cosine of an angle).

Returns

The angle, in radians, whose cosine is *x*. If *x* is not in the range -1.0 to 1.0, returns `NaN`.

Description

The arc cosine function (sometimes written as $\cos^{-1}$) is the inverse of the cosine function. It returns the angle whose cosine has the specified value, in radians. The return value is in the range of 0 to (i.e., 0 to 3.14159).

Example

```
trace (Math.acos(1.0));  // Displays: 0
trace (Math.acos(0.0));  // Displays: 1.5707... (i.e., pi/2)
```

See Also

Math.asin( ), Math.atan( ), Math.cos( )

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Chapter 18. ActionScript Language Reference

Math.asin( ) Method — Flash 5; can be used when exporting Flash 4 movies

compute the arc sine of a number

`Math.asin(`*x*`)`

Arguments

x

A number between -1.0 and 1.0 (the sine of an angle).

Returns

The angle, in radians, whose sine is *x*. If *x* is not in the range -1.0 to 1.0, returns `NaN`.

Description

The arc sine function (sometimes written as $\sin^{-1}$) is the inverse of the sine function. It returns the angle whose sine has the specified value, in radians. The return value is in the range of - /2 to /2 radians.

Example

```
trace (Math.asin(1.0));  // Displays: 1.5707...  (i.e., pi/2)
trace (Math.asin(0.0));  // Displays: 0
```

See Also

Math.acos( ), Math.atan( ), Math.sin( )

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Chapter 18. ActionScript Language Reference

Math.atan( ) Method — Flash 5; may be used when exporting Flash 4 movies

compute the arc tangent of a number

```
Math.atan(x)
```

Arguments

x

A number between `-Infinity` and `Infinity`, inclusive (the tangent of some angle).

Returns

The angle, in radians, whose tangent is *x*.

Description

The arc tangent function (sometimes written as $\tan^{-1}$) is the inverse of the tangent function. It returns the angle whose tangent has the specified value, in radians. The return value is in the range of - /2 to /2.

Example

```
trace (Math.atan(1.0));       // Displays: 0.78539...
trace (Math.atan(0.0));       // Displays: 0
trace (Math.atan(-Infinity)); // Displays: -1.5707... (i.e., -pi/2)
```

See Also

Math.acos( ), Math.asin( ), Math.tan( )

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Chapter 18. ActionScript Language Reference

Math.atan2( ) Method

Flash 5; may be used when exporting Flash 4 movies

---

determine an angle based on a point

```
Math.atan2(y, x)
```

Arguments

y

The y-coordinate of the point.

x

The x-coordinate of the point.

Returns

The angle, in radians, of the point (`x` , `y`) from the center of a circle, measured counterclockwise from the circle's positi horizontal axis (i.e., the X-axis). Ranges from to - . (Negative values indicate angles below the X-axis).

Description

The *atan2( )* method, like *atan( )* , performs an arc tangent calculation but uses x- and y-coordinates, rather than their ratio arguments. That is, calculating an arc tangent with *atan2( )* as:

```
Math.atan2(9, 3);  // Yields 1.24904577239825
```

is equivalent to calculating the arc tangent with *atan( )* , using the ratio of 9/3 (or 3), as follows:

```
Math.atan(3);      // Same thing
```

Usage

Note that the y-coordinate is passed as the first argument to *atan2( )* , whereas the x-coordinate is passed as the second arg This is intentional and required. It mirrors the structure of tangent, which is the ratio of the side opposite an angle (y) divi the side adjacent to the angle (x).

Example

The *atan2( )* method can be used to make a movie clip point toward a moving target. The following example, available at

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

online Code Depot, adds a custom method, *rotateTowardsMouse( )* , to all movie clips. It then assigns *rotateTowardsMou* the *onEnterFrame( )* handler of the clip `triangle_mc` , so that with every passing frame, the clip rotates towards the m pointer. The same technique could be used to orient an enemy spaceship toward a player's spaceship.

```
// Convert radians to degrees. There are 2*pi radians per 360 degrees.
_global.radiansToDegrees = function (radians) {
  return (radians/Math.PI) * 180;
}

// Method: MovieClip.rotateTowardsMouse()
//   Desc: Rotates a clip's positive x-axis towards the mouse pointer.
MovieClip.prototype.rotateTowardsMouse = function () {
  // Create a point object that stores the x- and y-coordinates of
  // this clip relative to its parent's registration point.
  var point = {x:this._x, y:this._y};
  // Convert the clip's parent's local coordinates to global (Stage) coordin
  this._parent.localToGlobal(point);
  // Measure the distance between the registration
  // point of this clip and the mouse.
  var deltaX = _root._xmouse - point.x;
  var deltaY = _root._ymouse - point.y;
  // Calculate the angle of the line from the registration point
  // of this clip to the mouse. Note that the y-coordinate is passed first.
  var rotationRadian = Math.atan2(deltaY, deltaX);
  // Convert the radian version of the angle to degrees.
  var rotationAngle = radiansToDegrees(rotationRadian); // See earlier funct
  // Update the rotation of this clip to point to the mouse.
  this._rotation = rotationAngle;
}

// Call rotateTowardsMouse() on every frame.
triangle_mc.onEnterFrame = rotateTowardsMouse;
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Chapter 18. ActionScript Language Reference

Math.ceil( ) Method

Flash 5; can be used when exporting Flash 4 movies

round a number up to the next integer

```
Math.ceil(x)
```

Arguments

x

A number.

Returns

The next integer greater than or equal to *x*.

Description

The *ceil( )* (i.e., ceiling) method converts a floating-point number to the first integer greater than or equal to *x*. It leaves integers unchanged.

Example

```
Math.ceil(1.00001);  // Returns 2
Math.ceil(5.5);      // Returns 6
Math.ceil(-5.5);     // Returns -5
Math.ceil(3);        // Returns 3
```

See Also

Math.floor( ), Math.round( )

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Chapter 18. ActionScript Language Reference

Math.cos( ) Method

Flash 5; can be used when exporting Flash 4 movies

---

compute the cosine of an angle

```
Math.cos(theta)
```

Arguments

theta

An angle, in radians (not degrees), normally in the range of 0 to 2 , although any angle will suffice, as cosine is a periodic function.

Returns

The cosine of `theta` (the result is in the range of -1.0 to 1.0).

Description

The *cos( )* function returns the trigonometric cosine of an angle. In a right triangle, the cosine of an angle is the result of dividing the length of the side adjacent to the angle by the triangle's hypotenuse.

Usage

Note that *cos( )* expects angles to be provided in radians, not degrees.

Example

```
trace (cos(0));                 // Displays: 1.0
trace (cos(Math.PI));           // Displays: -1.0
```

The *cos( )* function can be used along with *sin( )* to calculate a point on a circle, which we use in the following example to move a movie clip in a circular path. Given the radius of a circle, *r*, and an angle, xxxthetaxxx, measured counterclockwise from the positive horizontal axis, a point's location is (r*cosxxxthetaxxx, r*sinxxxthetaxxx):

```
// Global Function: degreesToRadians()
// Converts degrees to radians. There are 2*pi radians per 360 degrees.
_global.degreesToRadians = function (degrees) {
  return (degrees/180) * Math.PI;
}
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

```
// Method: MovieClip.moveInCircle()
//   Desc: Starts moving the clip around a circular path.
// Params: radius       Radius of circle path.
//         x            Horizontal center of circular path.
//         y            Vertical center of circular path.
//         degPerFrame  Number of degrees to move per frame rendered.
MovieClip.prototype.moveInCircle = function (radius, x, y, degPerFrame) {
  var deg = 0;          // Angle of object in degrees, measured
                        // counterclockwise from the horizon (X-axis).

  // Assign a movement function to run once per frame.
  this.onEnterFrame = function () {
    // Increase the rotation angle by 5 degrees
    deg += degPerFrame;

    // Place the object. Note that Flash inverts the Y-axis of Cartesian
    // coordinates so we decrement y to obtain our new location.
    var rotAngRad= degreesToRadians(deg);
    this._x = x + Math.cos(rotAngRad) * radius;
    this._y = y - Math.sin(rotAngRad) * radius;
  }
}

// Usage:
ball_mc.moveInCircle(100, 200, 200, 5);
```

See Also

Math.acos( ), Math.PI, Math.sin( ), Math.tan( )

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Chapter 18. ActionScript Language Reference

Math.E Property — **Flash 5; can be used when exporting Flash 4 movies**

the constant e (the base of the natural logarithm)

`Math.E`

Description

The `E` property stores an approximation of the natural logarithmic base (roughly 2.71828), which in mathematics is represented by the symbol *e*. It is a transcendental number, like , used in mathematical equations involving growth or change. Don't confuse it with the *E* that is used for exponential notation, as described in Section 4.3.2. The two are unrelated.

See Also

Math.log( ), `Math.LN10`, `Math.LN2`

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Chapter 18. ActionScript Language Reference

Math.exp( ) Method — Flash 5; can be used when exporting Flash 4 movies

raise the constant e to a specified power

```
Math.exp(x)
```

Arguments

x

The exponent to which to raise `Math.E`.

Returns

`Math.E` to the power *x* .

See Also

`Math.E`, `Math.LN10`, `Math.LN2`, *Math.log( )*, `Math.LOG10E`, *Math.pow( )*

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Math.floor( ) Method

Flash 5; can be used when exporting Flash 4 movies

round a number down to the previous integer

```
Math.floor(x)
```

Arguments

x

A number.

Returns

The closest integer less than or equal to *x*.

Description

The *floor( )* method converts a floating-point number to the first integer less than or equal to *x*. It leaves integers unchanged.

Example

```
Math.floor(1.99999);  // Returns 1
Math.floor(5.5);      // Returns 5
Math.floor(-5.5);     // Returns -6
Math.floor(3);        // Returns 3

function minutesToHHMM (minutes) {
  var hours = Math.floor(minutes/60);
  minutes -= hours * 60;
  minutes = minutes < 10 ? "0" + minutes : minutes;
  return hours + ":" + minutes;
}
```

See Also

Math.ceil( ), Math.round( )

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Chapter 18. ActionScript Language Reference

Math.LN10 Property — Flash 5; can be used when exporting Flash 4 movies

natural logarithm of 10 (loge10), approximately 2.30259

`Math.LN10`

Description

The `LN10` property represents the natural logarithm of 10 (the base- *e* logarithm of 10), a constant equaling approximately 2.30259.

See Also

Math.exp( ), `Math.LN2`, `Math.LOG10E`

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Chapter 18. ActionScript Language Reference

Math.LN2 Property — Flash 5; can be used when exporting Flash 4 movies

natural logarithm of 2 (loge2), approximately 0.69315

`Math.LN2`

Description

The `LN2` property represents the natural logarithm of 2 (the base -*e* logarithm of 2), a constant equaling approximately 0.69315

See Also

Math.exp( ), `Math.LN10`, `Math.LOG10E`

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Chapter 18. ActionScript Language Reference

Math.log( ) Method — Flash 5; can be used when exporting Flash 4 movies

compute the natural logarithm of a number

```
Math.log(x)
```

Arguments

x

A positive integer.

Returns

The natural logarithm of *x*.

Description

The *log( )* method calculates the natural logarithm (i.e., the base-*e* logarithm) of a number. See the following Example to calculate the base-10 logarithm.

Example

```
trace (Math.log(Math.E));   // Displays: 1

// Compute the base-10 logarithm of a number
function log10 (x) {
  return (Math.log(x) / Math.log(10));
}
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Chapter 18. ActionScript Language Reference

base-10 logarithm of e, approximately 0.43429

`Math.LOG10E`

Description

The `LOG10E` property represents the common logarithm of *e* (the base-10 logarithm of *e*), a constant equaling approximately 0.43429.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Chapter 18. ActionScript Language Reference

base-2 logarithm of e, approximately 1.44270

`Math.LOG2E`

Description

The `LOG2E` property represents the base-2 logarithm of *e* ($\log_2 e$), a constant equaling approximately 1.44270.

Bugs

In Flash Player 5.0.30.0, `LOG2E` erroneously returns the value of `Math.LN2` (0.69315). This is fixed in Flash 6.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Math.max( ) Method

Flash 5; can be used when exporting Flash 4 movies

---

determine the larger of two numbers

```
Math.max(x, y)
```

Arguments

x

A number.

y

A number.

Returns

The larger of `x` and `y` .

Description

The *max( )* function compares two numbers and returns the larger of the two. When comparing two negative numbers, the one with a smaller absolute value is actually larger (see the following Example). If used on strings, *max( )* returns `NaN` .

Example

```
Math.max(5, 1);                          // Returns 5
Math.max(-6, -5);                        // Returns -5
Math.max(Math.abs(-6), Math.abs(-5));  // Returns 6
```

This example constrains (clips) a value to the specified range:

```
function constrainToRange (checkVal, minVal, maxVal) {
  return Math.min(Math.max(checkVal, minVal), maxVal);
}
// Constrain the slider to between 100 and 200
mySlider._x = constainToRange(mySlider._x, 100, 200);
```

This example returns the maximum value in an array:

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

```
function maxInArray (checkArray) {
  maxVal = -Number.MAX_VALUE;  // Initialize maxVal to a very small number
  for (var i = 0; i < checkArray.length; i++) {
    maxVal = Math.max(checkArray[i], maxVal);
  }
  return maxVal;
}

trace(maxInArray([2,3,66,4,342,-90,0]));  // Displays: 342
```

See Also

Math.abs( ) , Math.min( )

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Chapter 18. ActionScript Language Reference

Math.min( ) Method — Flash 5; can be used when exporting Flash 4 movies

determine the smaller of two numbers

```
Math.min(x, y)
```

Arguments

x

A number.

y

A number.

Returns

The smaller of *x* and *y*.

Description

The *min( )* function compares two numbers and returns the smaller of the two. When comparing two negative numbers, the one with a larger absolute value is actually smaller (see the following Example). If used on strings, *min( )* returns `NaN`.

Example

```
Math.min(5, 1);                          // Returns 1
Math.min(-6, -5);                        // Returns -6
Math.min(Math.abs(-6), Math.abs(-5));  // Returns 5
```

Reader Exercise: Modify the example under *Math.max( )* to return the minimum value in an array rather than the maximum.

See Also

Math.abs( ), *Math.max( )*

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

the ratio of a circle's circumference to its diameter, approximately 3.14159

`Math.PI`

Description

The `PI` property represents the constant π, the ratio of the circumference of a circle to its diameter.

Example

`Math.PI` is famously used in calculating the area of a circle:

```
function circleArea (radius) {
  // PI times the radius squared can also be written
  // as Math.PI * Math.pow(radius, 2)
  return Math.PI * (radius * radius);
}
```

See Also

Math.cos( ), Math.sin( )

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Chapter 18. ActionScript Language Reference

Math.pow( ) Method — Flash 5; can be used when exporting Flash 4 movies

raise a number to a specified power

```
Math.pow(base, exponent)
```

Arguments

base

A number representing the base of the exponential expression.

exponent

A number representing the power (i.e., exponent) to which to raise `base`. If the base is negative, the exponent must be an integer, else `NaN` is returned.

Returns

The `base` raised to the power `exponent`.

Description

The *pow( )* method can be used to raise any number to any power. If `exponent` is negative, *pow( )* returns 1 / (`base`$^{\text{abs(}}$`exponent`$^{\text{)}}$). If `exponent` is a fraction, *pow( )* can be used to take the square root or cube root, and so on, of a number (in which case it returns the positive root, although, mathematically, there may also be a negative root).

Example

```
Math.pow(5, 2);    // Returns 25 (5 squared)
Math.pow(5, 3);    // Returns 125 (5 cubed)
Math.pow(5, -2);   // Returns 0.04 (1 divided by 25)
Math.pow(8, 1/3);  // Returns 2 (cube root of 8)
Math.pow(9, 1/2);  // Returns 3 (square root of 9)
Math.pow(10, 6);   // Returns 1000000 (can also be written as 1e6)
```

Bugs

Flash Player 5.0.30.0 did not correctly calculate *Math.pow( )* for negative values of `base`. For example, *Math.pow(-2, 2)* was calculated as `NaN`, whereas it should be 4. This was fixed in Flash Player 6.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

See Also

Math.exp( ), *Math.sqrt( )*, `Math.SQRT2`, `Math.SQRT1_2`

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Chapter 18. ActionScript Language Reference

Math.random( ) Method

Flash 5; can be used when exporting Flash 4 movies

generate a random number from 0 to 1.0

```
Math.random()
```

Returns

A floating-point number greater than or equal to 0.0 and less than 1.0.

Description

The *random( )* method provides a way to produce random numbers, which can then be used to choose randomly between actions in a script. The *random( )* method generates a random value between 0 and .99999..., inclusive, which we can scal according to our needs. For example, to obtain a random number between 0 and 5, we use:

```
Math.random() * 5
```

And to obtain a random integer between 1 and 6, we use:

```
Math.floor(Math.random() * 6) + 1
```

This custom function returns an integer number in a specified range rather than a floating-point number in the range 0 to 1

```
// Returns a number in the range of minVal to maxVal, inclusive
function myRandom (minVal, maxVal) {
  return minVal + Math.floor(Math.random() * (maxVal + 1 - minVal));
}

// Invoke the function
dieRoll = myRandom(1, 6);  // Emulates a six-sided die
trace(dieRoll);

// Note that to simulate two dice, you can use this:
twoDice = myRandom(2, 12);  // The minimum value is 2, not 1

// To return the die values separately, use an array
function rollTwoDice () {
  return [myRandom(1, 6), myRandom(1, 6)];
}
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Due to a bug in Flash Player 5.0.30.0, this approach is prone to an extremely rare, but potentially important inaccuracy. In Flash Player 5.0.30.0, *random( )* generates values in the range of 0.0 to 1.0, inclusive. When we multiply the return of *random( )* by an integer, *n* , we produce values in the range of 0.0 to *n* . In our example, we multiplied *Math.random( )* by so that the returned value ranges from 0.0 to 6.0. By invoking *floor( )* on the adjusted value, we produce integers in the ra of 0 to *n* (0 to 6 in our example). The scaling and rounding leads to an inaccurate distribution of random numbers—the chance of producing *n* is much smaller than the chance of producing any other number in the series.

The *Math.random( )* was fixed in Flash Player 5.0.41.0. If supporting older versions of the Player, the following version o the *myRandom( )* function avoids the problem by simply discarding the value 1.0 if it happens to be chosen by *Math.ran( )* :

```
// Returns an integer in the range of minVal to maxVal, inclusive
function myRandom (minVal, maxVal) {
  do {
    r = Math.random();   // Keep picking a number until it is not 1.
  } while (r =  = 1);
  return minVal + Math.floor(r * (maxVal + 1 - minVal));
}

// Invoke the function
dieRoll = myRandom(1, 6);  // Emulates a six-sided die safely in Flash 5.0.3
```

Usage

Math.random( ) replaces the deprecated Flash 4 *random* function.

Example

Math.random( ) is often used to cause the playhead to jump to a random frame in the timeline. The following code invoke *myRandom( )* function from the preceding example and then sends the playhead to the randomly chosen frame:

```
// Invoke the function; pick a random number between 10 and 20
var destinationFrame = myRandom(10, 20);

// Now send the playhead to that frame
gotoAndStop(destinationFrame);
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Chapter 18. ActionScript Language Reference

Math.round( ) Method

Flash 5; can be used when exporting Flash 4 movies

---

calculate the closest integer to a number

```
Math.round(x)
```

Arguments

x

A number.

Returns

The integer mathematically closest to *x* (or *x* itself, if *x* is an integer). If the fractional component of *x* is exactly 0.5 (*x* is equidistant from the two closest integers), *round( )* returns the first integer greater than *x*.

Description

The *round( )* method performs traditional rounding; it converts a floating-point number to the nearest integer. Positive numbers with a fractional portion *less* than 0.5, and negative numbers with a fractional portion *greater* than 0.5, are rounded *down*. Positive numbers with a fractional portion greater than or equal to 0.5, and negative numbers with a fractional portion *less* than or equal to 0.5, are rounded *up*.

Example

```
Math.round(1.4);      // Returns 1
Math.round(1.5);      // Returns 2
Math.round(1.6);      // Returns 2
Math.round(-5.4);     // Returns -5
Math.round(-5.5);     // Returns -5
Math.round(-5.6);     // Returns -6
```

See Also

int( ), Math.ceil( ), Math.floor( )

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Math.sin( ) Method

compute the sine of an angle

```
Math.sin(theta)
```

Arguments

theta

An angle, in radians (not degrees), normally in the range of 0 to 2 , although any angle will suffice, as sine is a periodic function.

Returns

The sine of `theta` (the result is in the range of -1.0 to 1.0).

Description

The *sin( )* method returns the trigonometric sine of an angle. In a right triangle, the sine of an angle is the result of dividing the length of the side opposite the angle by the triangle's hypotenuse.

Usage

Note that *sin( )* expects angles to be provided in radians, not degrees.

Example

```
trace(Math.sin(0));             // Displays: 0
trace(Math.sin(Math.PI/2));     // Displays: 1.0
```

The *sin( )* function can be used along with *cos( )* to calculate a point on a circle. See the Example under *Math.cos( )*.

See Also

Math.asin( ), *Math.cos( )*, *Math.PI*, *Math.tan( )*

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Math.sqrt( ) Method

Flash 5; can be used when exporting Flash 4 movies

---

calculate the square root of a number

```
Math.sqrt(x)
```

Arguments

x

A nonnegative integer.

Returns

The square root of `x`, or `NaN` if `x` is less than 0.

Description

The *sqrt( )* method returns the positive root of its operand (even though there may also be a mathematically valid negative root). It is equivalent to:

```
Math.pow(x, 0.5)
```

Example

```
Math.sqrt(4);    // Returns 2, although -2 is also a valid root
Math.sqrt(36);   // Returns 6, although -6 is also a valid root
Math.sqrt(-20);  // Returns NaN

// Calculate the distance between two points,
// the square root of (deltaX squared plus deltaY squared):
var dist = Math.sqrt((deltaX * deltaX) + (deltaY * deltaY));
```

See Also

Math.pow( ), `Math.SQRT2`, `Math.SQRT1_2`

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Chapter 18. ActionScript Language Reference

Math.SQRT1_2 Property — Flash 5; can be used when exporting Flash 4 movies

the reciprocal of the square root of 2, approximately 0.70711

`Math.SQRT1_2`

Description

The `SQRT1_2` property is a constant approximating the value `1/Math.SQRT2` (the reciprocal of the square root of 2), equaling approximately 0.70711.

See Also

`Math.SQRT2`

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Chapter 18. ActionScript Language Reference

Math.SQRT2 Property — Flash 5; can be used when exporting Flash 4 movies

the square root of 2, approximately 1.41421

`Math.SQRT2`

Description

The `SQRT2` property is a constant approximating the square root of 2, an irrational number equaling approximately 1.41421.

See Also

`Math.SQRT1_2`

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Chapter 18. ActionScript Language Reference

Math.tan( ) Method — Flash 5; can be used when exporting Flash 4 movies

compute the tangent of an angle

```
Math.tan(theta)
```

Arguments

theta

An angle, in radians (not degrees), normally in the range of - /2 to /2, although any angle will suffice, as tangent is a periodic function.

Returns

The tangent of `theta` (the result is in the range of `-Infinity` to `Infinity`).

Description

The *tan( )* method returns the trigonometric tangent of an angle. In a right triangle, the tangent of an angle is the result of dividing the length of the side opposite the angle by the length of the side adjacent to the angle. This is also the same as the ratio *Math.sin*`(theta)`/*Math.cos*`(theta)`, so as *cos*`(theta)` approaches zero, *tan*(`theta`) approaches `Infinity`. Therefore, *tan*`(theta)` is not calculable for the values - /2, /2, -3 /2, 3 /2, etc.

Example

```
trace (Math.tan(0));              // Displays: 0
trace (Math.tan(Math.PI/4));      // Displays: 1
```

See Also

Math.atan( ), Math.cos( ), Math.sin( )

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Chapter 18. ActionScript Language Reference

maxscroll Property Flash 6

the last legal top line of a text fieldFlash 4; replaced by TextField.maxscroll in

theTextField.maxscroll

Returns

A positive integer representing the line number of the last legal top line of a text field.

Description

The `maxscroll` property tells us the largest allowed `scroll` value for a text field. It represents the number of the last line in a text field that can be used as the top line in its viewable region. In Flash 4 and Flash 5, the `maxscroll` property is used with the `scroll` property to manage a scrolling-text field. As of Flash 6, `maxscroll` is formalized as a property of the *TextField* class, *`TextField`*`.maxscroll`.

See Also

`scroll`, *`TextField`*`.bottomScroll`, *`TextField`*`.hscroll`, *`TextField`*`.maxscroll`, *`TextField`*`.scroll`

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Chapter 18. ActionScript Language Reference

Mouse Object — **Flash 5; enhanced in Flash 6 to add listener events**

---

mouse events and control over mouse pointer visibility

`Mouse.methodName()`

Methods

addListener( )

Registers an object to receive *onMouseDown( )*, *onMouseMove( )*, and *onMouseUp( )* events.

hide( )

Hides the mouse pointer.

removeListener( )

Cancels event notices for the specified listener.

show( )

Enables the mouse pointer.

Listener events

onMouseDown( )

Occurs when the primary mouse button is depressed.

onMouseMove( )

Occurs when the mouse pointer moves.

onMouseUp( )

Occurs when the primary mouse button is released.

Description

The *Mouse* object provides feedback from mouse events and control over the visibility of the mouse pointer. Using the *Mouse* events, we can create mouse-driven interfaces that respond to mouseclicks and mouse movement. However, for menu- and button-based applications, the mouse-related *Button* events (*onPress( )*, *onRelease( )*, *onRollOver( )*, etc.) provide easier access to mouse actions. The mouse event handlers, such as *onMouseDown( )*, are useful when you want to trap mouseclicks not associated with a particular button. For example, you can use them to trap a mouseclick anywhere on-screen, which might be useful to advance a

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

slideshow.

We use *Mouse.hide( )* to conceal the system mouse pointer, usually in order to replace it with a customized mouse pointer, as shown in the Example under *Mouse.hide( )*. It may also be desirable to hide the mouse pointer in fullscreen, keyboard-controlled movies or for touch-screen kiosks.

Note that the *Mouse* object does not tell us the location of the mouse pointer. Use `_root._xmouse` and `_root._ymouse` to determine the mouse pointer's location relative to the main Stage. Use the `_xmouse` and `_ymouse` properties of a movie clip object to determine the mouse pointer's location relative to the clip.

Usage

The *onMouseDown( )*, *onMouseMove( )* and *onMouseUp( )* listener events were added to the *Mouse* object in Flash 6. They are generally preferred over the analogous Flash 5-style clip events *mouseDown*, *mouseMove*, and *mouseUp*, which are used within an *onClipEvent* block to detect mouse input. However, in Flash 6, the *updateAfterEvent( )* function, which refreshes the screen between frames, can be used only with *MovieClip* mouse or key events or with *setInterval( )*. Hence, when a screen refresh is required between frames, the *MovieClip.onMouseDown( )*, *MovieClip.onMouseMove( )*, and *MovieClip.onMouseUp( )* handlers must be used. See those entries in this *Language Reference* for details. Future versions of the Player will likely include support for *updateAfterEvent( )* from *Mouse* listener events.

See Also

Button.onPress( ), *Button.onRelease( )*, *MovieClip.onMouseDown( )*, *MovieClip.onMouseMove( )*, *MovieClip.onMouseUp( )*, `MovieClip._xmouse`, `MovieClip._ymouse`, *updateAfterEvent( )*

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Chapter 18. ActionScript Language Reference

Mouse.addListener( ) Method — Flash 6

register an object to receive onMouseDown( ), onMouseMove( ), and onMouseUp( ) events

```
Mouse.addListener(listener)
```

Arguments

listener

An object with *onMouseDown( )*, *onMouseMove( )*, and/or *onMouseUp( )* methods defined.

Description

The *addListener( )* method adds `listener` to the list of objects notified when *Mouse*'s *onMouseDown( )*, *onMouseMove( )*, and *onMouseUp( )* events occur. The `listener` parameter is an object of any class that defines methods by the names *onMouseDown*, *onMouseMove*, and/or *onMouseUp*; the methods are invoked when the corresponding event occurs for *Mouse*. Instances of built-in classes, such as *TextField* or *Button*, can respond to *Mouse* events. To stop a listener object from receiving events, use *Mouse.removeListener( )*.

Multiple objects can respond independently to each *Mouse* event. When multiple listeners are registered to *Mouse*, their methods are invoked in the order the listeners were added.

For general information on how listeners and callback functions work, see Chapter 10. For sample *Mouse* listener code, see *Mouse.onMouseDown( )*.

See Also

Mouse.onMouseDown( ), *Mouse.onMouseMove( )*, *Mouse.onMouseUp( )*, *Mouse.removeListener( )*; Chapter 10

About Safari | Terms of Service | Privacy Policy | Contact Us | Help | Submit a Problem

Copyright © 2002 O'Reilly & Associates, Inc. All rights reserved.
1005 Gravenstein Highway North
Sebastopol, CA 95472

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Chapter 18. ActionScript Language Reference

Mouse.hide( ) Method Flash 5

make the mouse pointer disappear

```
Mouse.hide()
```

Returns

Either 1 or 0, indicating whether the mouse was shown (1) or hidden (0) before the call to *hide( )*.

Description

The *hide( )* method causes the normal mouse pointer (usually an arrow, although sometimes a hand) to disappear when the mouse is over any part of the Flash Player. The normal system pointer reappears when the mouse passes outside the Flash Player's active Stage area.

Usage

Note that even after *Mouse.hide( )* has been invoked, the normal system text I-beam cursor appears when the mouse hovers over an editable text field.

Example

The *hide( )* method is used to conceal the default system mouse pointer, typically in order to replace it with a custom pointer. In Flash, a custom pointer is nothing more than a movie clip that follows the mouse. Using the *MovieClip.onMouseMove( )* event handler, we can cause a movie clip to follow the mouse by updating the clip's `_x` and `_y` properties when the mouse moves. The following code demonstrates the technique:

```
Mouse.hide();

// Makes the customPointer_mc movie clip follow the mouse.
// Note that customPointer_mc must be on the main timeline for this
// example to work. To place customPointer_mc on another timeline, we'd
// have to use globalToLocal() to convert from main timeline coordinates
// to the local customPointer_mc coordinate space.
customPointer_mc.onMouseMove = function () {
  this._x = _root._xmouse;
  this._y = _root._ymouse;
  // Refresh screen immediately, without waiting for the next frame.
  updateAfterEvent();
}
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

This code does not use a *Mouse.onMouseMove( )* listener, because in Flash 6 *updateAfterEvent( )* works only with a *MovieClip* mouse or key event or with *setInterval( )*.

It may also be desirable to hide the custom pointer when the mouse is inactive, say, because the pointer has left the Flash Player's active Stage area, in which case the system pointer appears, and there are two pointers on screen (the custom pointer and the system pointer). The following example shows how to adapt the preceding code to hide the cursor after 5 seconds of inactivity:

```
Mouse.hide();

// Makes the customPointer_mc movie clip follow the mouse.
customPointer_mc.onMouseMove = function () {
  this._x = _root._xmouse;
  this._y = _root._ymouse;
  this.lastMove = getTimer();
  // Refresh screen immediately, without waiting for the next frame.
  updateAfterEvent();
}

customPointer_mc.onEnterFrame = function () {
  if (getTimer() - this.lastMove > 5000) {
    this._visible = false;
  } else {
    this._visible = true;
  }
}
```

To change a custom pointer to a custom hand pointer when the mouse hovers over a button, use that button's *rollOver* event to set the custom pointer clip to a frame containing the custom hand, and use the *rollOut* event to set the custom pointer back to the default, as follows:

```
go_btn.onRollOver = function () {
  _root.customPointer_mc.gotoAndStop("hand");
}

go_btn.onRollOut = function () {
  _root.customPointer_mc.gotoAndStop("default");
}
```

In all cases, remember to place the custom pointer on the top layer of your movie so that it appears above all other content. Alternatively, use *duplicateMovieClip( )* or *attachMovie( )* to generate the custom pointer clip dynamically and assign it a very high depth.

See Also

Mouse.show( ), *MovieClip*._xmouse, *MovieClip*._ymouse

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Chapter 18. ActionScript Language Reference

Mouse.onMouseDown( ) Listener Event Flash 6

occurs when the primary mouse button is depressed

listener.onMouseDown( )

Description

Like the *Button.onPress ( )* event, the *onMouseDown ( )* event detects the downstroke of a mouseclick. Unlike the button *press* event, *onMouseDown ( )* is not tied to the *hit area* of a button object (a button's hit area is defined graphically when you create the button in the Flash authoring tool, whereas a movie clip's hit area can also be set via the `hitArea` property). The *onMouseDown ( )* event occurs each time the primary mouse button is depressed while the mouse pointer is over any part of the Stage.

To respond to an *onMouseDown ( )* event, we must assign a callback function to the `onMouseDown` property of some object, such as *`listener`* , and then add *`listener`* as a listener using *Mouse.addListener ( )* , as follows:

```
// Create a generic object (although it can be any type of object)
mouseListener = new Object();

// First assign a callback function to the object's onMouseDown property
mouseListener.onMouseDown = function () {
  trace("The mouse was pressed!");
}

// Then register the listener with the Mouse object
Mouse.addListener(mouseListener);
```

Any number of listener objects of any class can respond to the *onMouseDown ( )* event.

The *onMouseDown ( )* event executes immediately when the mouse is pressed, independently of the code on a movie's timeline(s). However, visual changes to the movie made by the listener are not rendered until the next frame refresh, which is determined by the frame rate of the movie. Unfortunately, in Flash 6, *updateAfterEvent ( )* cannot be used to force a screen refresh from within an *onMouseDown ( )* listener; *updateAfterEvent ( )* has an effect only when invoked either from a movie clip mouse or key event or a *setInterval ( )* callback. For example:

```
someMovieClip_mc.onMouseDown = function () {
  // Scale the clip to 200% of its original size.
  this._xscale = this._yscale = 200;
  // Refresh the screen. In Flash 6, this does not work in a mouse listener.
  updateAfterEvent();
}
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

See Also

Button.onPress( ) , Mouse.addListener( ) , Mouse.onMouseMove( ) , Mouse.onMouseUp( ) , MovieClip.onMouseDown( )

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Chapter 18. ActionScript Language Reference

Mouse.onMouseMove( ) Listener Event — Flash 6

occurs when the mouse pointer moves

`listener.onMouseMove()`

Description

The *onMouseMove( )* event lets us detect changes in the mouse pointer's position. Whenever the mouse is in motion, *onMouseMove( )* events are issued repeatedly, as fast as the processor can generate new events. Each time *onMouseMove( )* executes, we know the mouse pointer has changed position. We can check that position relative to any movie clip by querying the clip's `_xmouse` and `_ymouse` properties. To check the position of the mouse pointer relative to the main Stage, use `_root._xmouse` and `_root._ymouse`.

To respond to an *onMouseMove( )* event, we must assign a callback function to the `onMouseMove` property of some object, such as *`listener`*, and then add *`listener`* as a listener using *Mouse.addListener( )*, as follows:

```
// Create a generic object (although it can be any type of object)
mouseListener = new Object();

// Assign a callback function to the object's onMouseMove property
mouseListener.onMouseMove = function () {
  trace("The mouse was moved!");
}

// Register the listener with the Mouse object
Mouse.addListener(mouseListener);
```

Any number of listener objects of any class can respond to the *onMouseMove( )* event.

The *onMouseMove( )* event is useful for code that wakes up idle applications, displays mouse trails, or creates custom pointers. However, for visual applications in Flash 6, the *MovieClip.onMouseMove( )* handler is preferred over the *Mouse.onMouseMove( )* listener event because the former allows for between-frame screen refreshes.

The *onMouseMove( )* event executes immediately when the mouse moves, independently of the code on a movie's timeline(s). However, visual changes to the movie made by the listener are not rendered until the next frame refresh, which is determined by the frame rate of the movie. Unfortunately, in Flash 6, *updateAfterEvent( )* cannot be used to force a screen refresh from within an *onMouseMove( )* listener; *updateAfterEvent( )* has an effect only when invoked either from a movie clip mouse or key event or a *setInterval( )* callback. Hence, in Flash 6, it's often appropriate to create a movie clip simply to implement *onMouseMove( )* with a screen refresh. For example:

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

```
// Create the movie clip
this.createEmptyMovieClip("mouseListener_mc", 1);
// Assign a callback function to the clip's onMouseMove handler
mouseListener_mc.onMouseMove = function () {
  // Respond to mouse movement (details omitted)...then refresh screen
  updateAfterEvent();
}
```

See the Example under *Mouse.hide*( ) for custom pointer code. Mouse trailer code is available at the online Code Depot, in varying degrees of complexity.

See Also

Mouse.addListener( ), *Mouse.onMouseDown( )*, *Mouse.onMouseUp( )*, *MovieClip.onMouseMove( )*

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Chapter 18. ActionScript Language Reference

Mouse.onMouseUp( ) Listener Event — Flash 6

occurs when the primary mouse button is released

```
listener.onMouseUp()
```

Description

The *onMouseUp( )* event is the counterpart of *onMouseDown( )*. It occurs each time the primary mouse button is pressed (while the mouse pointer is over any part of the Stage) and then released.

To respond to an *onMouseUp( )* event, we must assign a callback function to the `onMouseUp` property of some object, such as *`listener`*, and then add *`listener`* as a listener using *Mouse.addListener( )*, as follows:

```
// Create a generic object (although it can be any type of object)
mouseListener = new Object();

// Assign a callback function to the object's onMouseUp property
mouseListener.onMouseUp = function () {
  trace("The mouse was released!");
}

// Register the listener with the Mouse object
Mouse.addListener(mouseListener);
```

Any number of listener objects of any class can respond to the *onMouseUp( )* event.

The *onMouseUp( )* event executes immediately when the mouse is released, independently of the code on a movie's timeline(s). However, visual changes to the movie made by the listener are not rendered until the next frame refresh, which is determined by the frame rate of the movie. Unfortunately, in Flash 6, *updateAfterEvent( )* cannot be used to force a screen refresh from within an *onMouseUp( )* listener; *updateAfterEvent( )* has an effect only when invoked either from a movie clip mouse or key event or a *setInterval( )* callback.

See Also

Button.onRelease( ), *Mouse.addListener( )*, *Mouse.onMouseDown( )*, *Mouse.onMouseMove( )*, *MovieClip.onMouseUp( )*

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Chapter 18. ActionScript Language Reference

Mouse.show( ) Method Flash 5

make the mouse pointer appear

```
Mouse.show( )
```

Returns

Either 1 or 0, indicating whether the mouse was shown (1) or hidden (0) before the call to *show( )*.

Description

The *show( )* method causes the default system mouse pointer to reappear after *hide( )* has been invoked. It can be used to provide a normal pointer at necessary times in a movie, such as when the user is expected to fill in a form.

See Also

Mouse.hide( )

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Chapter 18. ActionScript Language Reference

Mouse.removeListener( ) Method Flash 6

cancels event notices for the specified listener

```
Mouse.removeListener(listener)
```

Arguments

listener

An object that was previously registered as a listener for *Mouse*.

Returns

A Boolean; `true` if `listener` was found and removed, `false` if `listener` was not registered as an event listener for *Mouse*. The return value is useful for debugging code that removes listeners.

Description

The *removeListener( )* method stops `listener` from receiving *Mouse* event notifications. It should only be used after `listener` has been registered as an event listener via *addListener( )*. The listener object is not deleted, but it can be removed and added repeatedly. Event listeners should be removed when the task they perform is no longer required.

See Also

Mouse.addListener( ), *Mouse.onMouseDown( )*, *Mouse.onMouseMove( )*, *Mouse.onMouseUp( )*; Chapter 10

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Chapter 18. ActionScript Language Reference

MovieClip Class Flash 6

class-like datatype for main movies and movie clipsFlash 3; enhanced in Flash 4, Flash 5, and

Constructor

```
new MovieClip();
```

The constructor syntax is used only to create *MovieClip* subclasses. It does not create an on-stage movie clip! To create a movie clip instance, use *attachMovie( )*, *createEmptyMovieClip( )*, or *duplicateMovieClip( )*. *MovieClip* symbols, on which instances are based, must be created manually in the Flash authoring tool.

Properties

Movie clip properties provide descriptions of, and control over, various physical features of movie clips and main movies. As with any object, *MovieClip* properties are accessed using the dot or [ ] operators. See Section 13.1 for details on using movie clip properties.

Here are three property-related issues to note:

- Changing a physical property of a *MovieClip* object places that clip under programmatic control, breaking the internal timeline's hold on it. This halts any tween currently in progress.
- Some *MovieClip* properties appear to be floating-point numbers but are mapped to other formats internally. The `_alpha` property, for example, is mapped internally to an integer between 0 and 255. This results in a loss of precision between setting and retrieving values of certain properties. For example, we may set a property value to 90 and then immediately retrieve the value back as 89.84375. If minor variations in property values make a difference to a script, we must manually round those values using *Math.round( )*, *Math.floor( )*, or *Math.ceil( )* after retrieval, or we must store the original value in a variable separate from the property.
- The built-in *MovieClip* properties are not enumerated by a *for-in* loop.

Table 18-11 lists the properties of the *MovieClip* class. Movie clip properties can be accessed for any movie clip instance and, in most cases, on the main movie timeline (`_root`). Those that cannot be used with the main movie timeline are marked by [5] in Table 18-11.

Prior to Flash 6, all built-in movie clip properties were prefaced with an underscore. To conform more closely with other ECMA-262 languages, properties that are new in Flash 6, and are not analogous to legacy Flash 5 *Button* properties such as `_focusRect`, do not obey the underscore convention. Properties introduced in

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Flash 6 are marked by [6] character in Table 18-11.

In the *Access* column of Table 18-11, *R/W* indicates that a property's value can be both retrieved and set (i.e., read/write), and *RO* indicates that it can be retrieved but not set (i.e., read-only). Some read-only properties can sometimes be set indirectly through the authoring tool or via some related function (e.g., *gotoAndStop( )* sets the `_currentframe` property), but only read/write properties can be set directly via ActionScript. The *Type* column describes the datatype of each property's value. The *Property description* column gives a quick summary of the property's purpose, but the full descriptions that follow later provide important details. All properties of the current timeline can be accessed without an explicit reference to it, as in `_alpha` versus *`someClip`*`._alpha`.

Table 18-11. Movie clip property summary

Property name	Access	Type	Property description
`_alpha`	R/W	number	Opacity percentage: 0 is transparent; 100 is opaque
`_currentframe`	RO	number	Frame number at which playhead resides
`_droptarget`	RO	string	Target path of the clip over which a dragged clip hovers or has been dropped, in slash notation
`enabled`	R/W	Boolean	Allows or disallows button interaction
`focusEnabled`	R/W	Boolean	Allows the clip to receive keyboard focus programmatically
`_focusrect`	R/W	Boolean	Disables automatic yellow rectangle around focused movie clips
`_framesloaded`	RO	number	Number of frames that have been downloaded
`_height`	R/W	number	Height, in pixels, of the current contents
`hitArea`[5][6]	R/W	string	Assigns the clickable area for a button-enabled movie clip
`_name`[5]	R/W	string	Identifier of an instance as a string (not a reference)
`_parent`[5]	RO	MovieClip reference	A reference to the instance or movie that contains the current instance
`_rotation`	R/W	number	Degrees of rotation
`tabChildren`[5][6]	R/W	Boolean	Controls whether contained objects are included in the automatic tab order
`tabEnabled`[5][6]	R/W	Boolean	Includes or excludes the movie clip from the current tab order
`tabIndex`[5][6]	R/W	Boolean	Specifies the movie clip's index in the custom tab order
`_target`	RO	string	Target path in absolute terms, in slash notation
`_totalframes`	RO	number	Number of frames in the timeline
`trackAsMenu`[5][6]	R/W		Modifies the *onRelease( )* handler requirements, enabling menu-style behavior
`_url`	RO	string	Disk or network location of the source *.swf* file
`useHandCursor`[5][6]	R/W	Boolean	Dictates whether a hand cursor is displayed when the mouse is over the clip (applies to clips with button handlers only)
`_visible`	R/W	Boolean	Visibility: `true` if shown; `false` if hidden
`_width`	R/W	number	Width, in pixels, of the current contents
`_x`	R/W	number	Horizontal position, in pixels

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

_xmouse	RO	number	Horizontal location of mouse pointer, in pixels
_xscale	R/W	number	Horizontal scaling percentage
_y	R/W	number	Vertical position, in pixels
_ymouse	RO	number	Vertical location of mouse pointer, in pixels
_yscale	R/W	number	Vertical scaling percentage

[5] Applies to movie clip instances only; does not apply to the main timeline (_root).

[6] New in Flash 6. Not supported in prior versions.

Methods

Table 18-12 and Table 18-13 list the methods of the *MovieClip* class.

Movie clip methods can be invoked on any movie clip instance and, in most cases, on the main movie timeline (_root). Those in Table 18-12 that cannot be used with the main movie timeline are marked by [8]. Methods introduced in Flash 6 are marked by [7] in the tables. The movie clip methods marked by [9] character in Table 18-12 provide the same functionality as analogous global functions but use the preferred *MovieClip.method( )* dot syntax format (that is, a movie clip instance must precede the dot). Methods that do not correspond to a global function can be applied to the current clip without an explicit reference to the movie clip instance, as in *attachMovie( )* versus *someClip.attachMovie( )*.

Table 18-12. Movie clip method summary

Method name	Method description
attachMovie( )	Creates a new instance based on an exported symbol from the current document's Library
createEmptyMovieClip( )[7]	Creates a new, empty *MovieClip* instance
createTextField( )[7]	Creates a new *TextField* object
duplicateMovieClip( )[8][9]	Creates a copy of a movie clip instance
getBounds( )	Returns an object whose properties give the coordinates of the bounding box that defines the visual region occupied by the clip
getBytesLoaded( )	Returns the number of downloaded bytes of an instance or a movie (not applicable to internal clips)
getBytesTotal( )	Returns the physical byte size of an instance or a main movie
getDepth( )[7]	Returns the movie clip's position in the visual content stack
getURL( )[9]	Loads an external document (usually a web page) into the browser
globalToLocal( )	Converts the properties of a *coordinates* object from Stage coordinates to instance coordinates
gotoAndPlay( )[9]	Moves the playhead of an instance or movie to a specific frame and then plays the instance or movie
gotoAndStop( )[9]	Moves the playhead of an instance or movie to a specific frame and then stops the playhead

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

hitTest( )	Returns a Boolean indicating whether a clip intersects with a given point or another clip
loadMovie( )[9]	Loads an external *.swf* file into the Player
loadVariables( )[9]	Retrieves external data composed of variable names and values, and converts that data into equivalent ActionScript variables
localToGlobal( )	Converts the properties of a *coordinates* object from an instance's coordinates to Stage coordinates
nextFrame( )[9]	Moves the playhead of an instance or movie ahead one frame and stops it there
play( )[9]	Starts the playhead of an instance or movie in motion (i.e., plays the clip)
prevFrame( )[9]	Moves the playhead of an instance or movie back one frame and stops it there
removeMovieClip( )[8][9]	Deletes a duplicated or attached instance
setMask( )[7]	Assigns a movie clip as a mask for another clip
startDrag( )[9]	Causes an instance or movie to follow the mouse pointer physically
stop( )[9]	Halts the playback of an instance or movie
stopDrag( )[9]	Ends any drag operation in progress
swapDepths( )[8]	Alters the position of an instance in the content stack
unloadMovie( )[9]	Removes an instance or main movie from a document level or host clip
valueOf( )	Returns an object reference to the instance

[7] New in Flash 6. Not supported in prior versions.

[8] Applies to movie clip instances only; does not apply to the main timeline (`_root`).

[9] MovieClip method that is analogous to a global function of the same name.

Methods to draw lines and fills in a movie clip instance, the so-called *Drawing API*, are also new in Flash 6 and are shown in Table 18-13. Refer also to the *MovieClip.createEmptyMovieClip( )*, method, which is convenient for creating an empty canvas on which to draw.

Table 18-13. Movie clip Drawing API methods

Method name	Method description
beginFill( )[10]	Starts a solid-filled shape
beginGradientFill( )[10]	Starts a gradient-filled shape
clear( )[10]	Removes all drawings from the movie clip
curveTo( )[10]	Draws a curved line
endFill( )[10]	Finishes a filled shape
lineStyle( )[10]	Sets stroke thickness, color, and transparency
lineTo( )[10]	Draws a straight line
moveTo( )[10]	Moves the drawing pen to a new position without drawing a line

[10] New in Flash 6. Not supported in prior versions.

Events

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Movie clip instances support *event handlers*, which respond automatically to a predefined set of events (e.g., mouse and keyboard interaction or movie playback). As of Flash 6, *Button* events (e.g., *onPress( )*, *onRelease( )*, and *onRollOver( )*) are also available for movie clips, although not for main movie timelines (`_root`). Movie clip event handlers can be created in the following ways:

- By attaching an *onClipEvent( )* block directly to a movie clip instance at authoring time
- By attaching a button-style *on( )* block directly to a movie clip instance at authoring time
- By assigning a callback function to the associated event property with ActionScript

Here are examples of each of these techniques:

```
// An onClipEvent() block
onClipEvent (enterFrame) {
  // statements
}

// An on() block
on (release) {
  // statements
}

// Two callback functions
theClip_mc.onEnterFrame = function () {
  // statements
}

theClip_mc.onPress = function () {
  // statements
}
```

Callback functions are not supported by Flash 5 but are generally preferred over *onClipEvent( )* and *on( )* blocks as of Flash 6. For important details on the differences between the types of event handlers, see Chapter 10. In particular, note that:

- When the contents of a movie clip are unloaded, event handler properties (e.g., *onEnterFrame( )*) are lost, but *onClipEvent( )* handlers (e.g., *onClipEvent(enterFrame)*) are not.
- When a movie clip is duplicated via *duplicateMovieClip( )*, the event handler properties are not copied to the duplicate clip, but the *onClipEvent( )* handlers are.

The supported movie clip event handlers are listed in Table 18-14. Handlers listed with [11] are not supported by main movies (`_root`). All the event handlers in Table 18-14 are new in Flash 6, but the Flash 5 equivalents are shown where applicable. With the exception of *onKillFocus( )* and *onSetFocus( )*, events listed without Flash 5 equivalents are adopted *Button* events.

Table 18-14. Movie clip event handler summary

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Clip event handler	Clip event occurs when...	Flash 5 equivalent
onData( )	The clip receives the end of a stream of loaded variables, or a portion of a loaded movie	*onClipEvent(data)*
onDragOut( )[11]	The mouse is pressed over the clip, then the mouse moves off the clip before the mouse is released	None
onDragOver( )[11]	The mouse is pressed over the clip, then the mouse moves off and back onto the clip	None
onEnterFrame( )	One tick of the frame rate passes in the Flash Player	*onClipEvent(enterFrame)*
onKeyDown( )	A key is pressed down while the clip is on stage and has keyboard focus	*onClipEvent(keyDown)*
onKeyUp( )	A depressed key is released while the clip is on stage and has keyboard focus	*onClipEvent(keyUp)*
onKillFocus( )[11]	The clip loses keyboard focus	None
onLoad( )[11]	The clip first appears on stage, or a *.swf* file finishes loading into the clip	*onClipEvent(load)*
onMouseDown( )	Primary mouse button is depressed while the clip is on stage	*onClipEvent(mouseDown)*
onMouseMove( )	Primary mouse button is depressed and then released while the clip is on stage	*onClipEvent(mouseMove)*
onMouseUp( )	Mouse pointer moves (even a teensy bit) while the clip is on stage	*onClipEvent(mouseUp)*
onPress( )[11]	The mouse is pressed over the clip	None
onRelease( )	The mouse is pressed and then released over the clip	None
onReleaseOutside( )[11]	The mouse is pressed over the clip and then released after moving off the clip	None
onRollOut( )[11]	The mouse pointer moves off the clip while the mouse button is not depressed	None
onRollOver( )[11]	The mouse pointer moves over the clip while the mouse button is not depressed	None
onSetFocus( )[11]	The clip gains keyboard focus	None
onUnload( )[11]	The clip is removed from the Stage	*onClipEvent(unload)*

[11] Applies to movie clip instances only; does not apply to the main timeline (`_root`).

Description

MovieClip is not just a class in ActionScript; it is also a unique ActionScript datatype used to represent information about, and allow control of, movies and movie clips. For most purposes, we treat movie clips and movies as objects; we can create and access movie clip properties, and we can create and invoke movie clip methods. For a full consideration of the *MovieClip* class, see Chapter 13.

We do not use a constructor to create new *MovieClip* instances. Instead, we create movie clip symbols in the authoring tool and place instances on the Stage manually. Some methods, however, allow us to copy existing clips (e.g., *duplicateMovieClip( )*) or add new clips to the Stage programmatically (e.g., *attachMovie( )* and *createEmptyMovieClip( )*).

Not all *MovieClip* objects are equal; some *MovieClip* methods and properties apply to movie clip instances only, not to main movies (a main movie is the `_root` timeline of a *.swf* document). In our consideration of the

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

MovieClip properties and methods, we'll note cases in which functionality is limited to one type of *MovieClip* object. Note that we use the word *MovieClip* to refer to the "class" of the objects, and the word *movieclip* (lowercase) to refer to the ActionScript datatype; we use the words *movie clip*, *clip*, or *instance* to refer to a particular movie clip, and we use *movie* to refer to the main movie of a *.swf* file. In the synopsis for each detailed property and method entry, the abbreviation `mc` stands in for the name of a clip or main movie. For many properties and methods, `mc` is optional—if omitted, the current timeline is assumed (see Table 18-12).

Throughout this *MovieClip* section, when talking about coordinates, we need to refer to the location of movie clips. We measure the position of a clip in reference to one representative point, its so-called *registration point*, as marked by a crosshair in the clip's Library symbol.

When a clip resides on the main Stage, we describe its location relative to the top-left corner of the Stage, which corresponds to the point (0,0). When a clip resides within another clip, we describe its location relative to the registration point of the parent clip, which again corresponds to the point (0,0). The point (0,0) in both cases is the origin point (or simply *origin*) of the coordinate space being used to plot the location of the clip. We'll see how the *localToGlobal( )* and *globalToLocal( )* methods can convert between these two coordinate spaces.

Flash's coordinate system inverts the Y-axis of Cartesian coordinates; that is, y values increase in a downward direction, not upward. For example, a y-coordinate of 100 indicates a point 100 pixels *below* the X-axis.

We'll often use the coordinate-related properties and methods to move a clip, determine where a clip is, or determine whether it intersects another object or point. The last technique is referred to as *collision detection* because it is often used to determine whether to change the direction in which a clip is animating, as if it bounced off another object (see the *MovieClip.hitTest( )* method).

Note that ActionScript doesn't have a native datatype to represent a *point* (i.e., an x- and y-coordinate). See *MovieClip.globalToLocal( )* for an explanation of how to create a point object from a generic object.

In Flash 5, no direct API was available to create ActionScript subclasses of the *MovieClip* class. As of Flash 6, the *Object.registerClass( )* method can be used to register a custom class as a subclass of *MovieClip*. For full details, see Chapter 14.

As of Flash 6, movie clip instances can behave exactly like buttons. To implement button behaviors for a movie clip, define one or more of the following button event handlers: *onDragOut( )*, *onDragOver( )*, *onPress( )*, *onRelease( )*, *onReleaseOutside( )*, *onRollOut( )*, *onRollOver( )* (see Table 18-14). Once a button handler is defined for a movie clip, the following button-related properties become active: `enabled`, `hitArea`, `trackAsMenu`, and `useHandCursor`. These properties can be set at any time, but they have an effect only when at least one button handler is defined. To provide button-style Up, Over, and Down states for a movie clip, create frames with the special labels "`_up`", "`_over`", and "`_down`". To define the button-style Hit state for a movie clip, use the `hitArea` property. If no `hitArea` property is defined, the clip's own content is used as the hit area. Note that though Macromedia's documentation refers to movie clips with button handlers as "button movie clips," there is no such symbol type or class. A "button movie clip" is simply Macromedia's term for a movie clip that happens to define at least one button handler. See Section 13.9.

See Also

Chapter 13, Chapter 14

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Chapter 18. ActionScript Language Reference

MovieClip._alpha Property — Flash 4

opacity of a clip or movie — *read/write*

mc._alpha

Description

The floating-point _alpha property specifies the opacity (or, conversely, the transparency) of *mc* as a percentage—0 is completely transparent, whereas 100 is completely opaque. Setting the _alpha property of *mc* affects the visual transparency of all clips nested inside *mc* but does not affect their _alpha properties. That is, if we have a clip, square, that contains another clip, circle, and we set square._alpha to 50, then circle will be 50% transparent on-screen but will have an _alpha of 100. However, if both circle and its parent clip (square) set their _alpha properties, the effect is cumulative. If each have an _alpha of 50, circle will appear 25% opaque (75% transparent) on-screen. Therefore, don't set the _alpha of nested clips unless they are supposed to have different transparencies from any siblings. To keep it simple, set the _alpha of the parent clip instead.

Usage

Note that setting the _alpha of a movie clip affects the aa property of the object returned by *Color.getTransform( )*.

Example

```
ball_mc._alpha = 60;   // Make ball_mc partially transparent
ball_mc._alpha = 0;    // Make ball_mc invisible
```

The following clip event handler makes theClip_mc more transparent as the mouse moves down the screen:

```
theClip_mc.onEnterFrame = function () {
  this._alpha = 100 - (_root._ymouse / 400) * 100;
}
```

See Also

The *Color* class; *MovieClip*._visible

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

MovieClip.attachMovie( ) Method Flash 6

create a new movie clip instance from an exported Library symbolFlash 5; initObject and return value added in

```
mc.attachMovie(symbolIdentifier, newName, depth)
mc.attachMovie(symbolIdentifier, newName, depth, initObject)
```

Arguments

symbolIdentifier

A string specifying the linkage identifier of an exported movie clip symbol, as set in the Library under Options

newName

A string specifying the new instance name of the clip being created. The name must adhere to the rules for creating identifier, outlined under Section 15.5.

depth

An integer between -16384 and 1048575 (inclusive), specifying the level on which to place the new clip inside *mc* depths obscure lower depths, so a depth of 6 is in front of 5, which is in front of 4, and so on. Depths above -1 are for dynamically generated content. Therefore, in most cases, you should use a depth between 0 and 1048575 with *attachMovie( )* . However, when creating dynamic content that you want to appear beneath all author-time content (e.g., a run-time background), use depth -16384. Depths from -16383 to -1 (inclusive) are reserved for author-time and shouldn't be used with *attachMovie( )* . For important details, see Section 13.4.

initObject

An optional object whose properties are copied to the new movie clip instance. Not supported in Flash 5.

Returns

In Flash 6, *attachMovie( )* returns an object reference to the attached movie clip, or `undefined` if creation fails (in Flash always returns `undefined`).

Description

The *attachMovie ( )* method creates a new instance, called *`newName`* , based on the exported movie clip symbol specified *`symbolIdentifier`* . If *`mc`* is the main movie, the new instance is placed in the top-left corner of the Stage. If *`mc`* clip instance, the new instance is placed at *`mc`* 's registration point. In either case, the new instance is placed inside *`mc`* 's content stack on the level specified by *`depth`* . For example, the following code attaches a movie clip exported from the with a linkage identifier of `"ball_symbol"` . It names the new instance `tennisBall_mc` and places it inside `court_mc` on depth 1, above all author-time content in `court_mc` :

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

```
court_mc.attachMovie("ball_symbol", "tennisBall_mc", 1);
```

We can then control the new instance as follows:

```
// Double the ball's width
court_mc.tennisBall_mc._xscale = 200;
```

As of Flash 6, *attachMovie ( )* returns a reference to the new instance, so this code can be shortened to:

```
tBall = court_mc.attachMovie("ball_symbol", "tennisBall_mc", 1);
tBall_mc._xscale = 200;
```

or, even more succinctly (but cryptically):

```
court_mc.attachMovie("ball_symbol", "tennisBall_mc", 1)._xscale = 200;
```

To delete an attached instance, use *MovieClip.removeMovieClip ( )* (see the global *removeMovieClip( )* function descript further important details).

In Flash 5, all exported movie clips are loaded before the first frame of the movie, causing a delay before the movie loads. Flash 6, exported clips can be loaded on other frames as follows:

1. In the Library's upper-right corner, select Linkage from the pop-up Options menu.
2. In the Linkage Properties dialog box, uncheck "Export in first frame," and then choose OK.
3. On the main timeline, at the frame where the exported clip should load, create a keyframe and place an instance of t it.

A movie clip is not available to *attachMovie ( )* until the frame at which it is exported.

When the optional *initObject* parameter is provided, its *direct properties* (properties attached directly to *initObj* those retrieved via the prototype chain) are copied into the new clip and become immediately accessible. For example:

```
// Create a generic object to transfer properties to the new tennisBall_mc i
var tempObj = new Object();
// Add a speed property for transfer to tennisBall_mc
tempObj.speed = 200;
// Pass tempObj to attachMovie()
court_mc.attachMovie("ball_symbol", "tennisBall_mc", 1, tempObj);
// Check the value of speed on tennisBall_mc
trace(court_mc.tennisBall_mc.speed);  // Displays: 200
```

However, properties inherited through *initObject* 's prototype chain are not copied to the clip. Only direct propertie transferred. To determine whether a property is inherited or direct, use *Object.hasOwnProperty ( )* . Built-in movie clip pr such as _x and _rotation, and event handlers, such as *onEnterFrame ( )* , can be set (i.e., copied to the newly create clip instance) via *initObject* . For example:

```
var tempObj = new Object();
// Set _rotation
tempObj._rotation = 200;
// Assign an onEnterFrame() callback
tempObj.onEnterFrame = function () {
  this._x++
};
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

```
// Once attached, tennisBall_mc will be rotated by 200 degrees and will
// start moving one pixel right per frame due to the onEnterFrame() handler
court_mc.attachMovie("ball_symbol", "tennisBall_mc", 1, tempObj);
```

In Flash 6, the *initObject* parameter is used to set properties in *newName* that will be used by its class constructo (effectively the same as passing arguments to the constructor). A movie clip symbol's class constructor is defined by the *Object.registerClass ( )* method. For convenience, *initObject* can be specified as an anonymous object. For example

```
court_mc.attachMovie("ball_symbol", "tennisBall_mc", 1, {speed: 200, _x: 165
```

The *initObject* parameter is not supported prior to Flash 6.

See Also

duplicateMovieClip( ) , *MovieClip.createEmptyMovieClip( )* , *MovieClip.duplicateMovieClip* ( ); Section 13.3.2.3 and S 13.4.3.1; Chapter 14

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

MovieClip.beginFill( ) Method Flash 6

starts drawing a solid-filled shape

```
mc.beginFill(RGB, alpha)
```

Arguments

RGB

An integer specifying the color of the shape, normally supplied in RGB hexadecimal triplet notation—`0x` *RRGGB* where *RR* , *GG* , and *BB* are two-digit hex numbers representing Red, Green, and Blue. For example, the hex int `0xFF0000` indicates red. If *RGB* is omitted or `undefined` , no fill is drawn. For a complete discussion of RG notation, see the *Color* class.

alpha

An integer between 0 and 100 specifying the opacity (or, conversely, the transparency) of the shape as a percentage completely transparent, whereas 100 is completely opaque. Defaults to 100. Values above 100 are interpreted as 10 below 0 are interpreted as 0.

Description

Part of the Drawing API, the *beginFill ( )* method starts drawing a new shape within *mc* , filled with a solid color of *RGB* transparency of *alpha* . The shape appears *above* any existing Drawing API content in *mc* but *beneath* all other conten (e.g., movie clips, shapes, text, and buttons placed at authoring time, or content attached at runtime via *attachMovie ( )* , *duplicateMovieClip ( )* , or *createTextField ( )*). To draw a shape with a gradient fill, use *beginGradientFill ( )* .

The *beginFill ( )* method is only one component of the shape drawing process. Think of the Drawing API as an old line pl (like the kind used for architectural drawings) with an imaginary *pen* that can be lifted, dropped, or moved in order to draw and curves. To draw a filled shape, we must combine *beginFill ( )* with other *MovieClip* drawing methods, as follows:

1. Position the drawing pen with *moveTo( )* .
2. Start the shape with *beginFill( )* .
3. Draw the shape's outline with a series of *lineTo( )* and/or *curveTo( )* calls, the last of which must end at the point sp in the original *moveTo( )* call.
4. Close the shape with *endFill( )* .

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

You can draw into an existing or empty movie clip with the Drawing API. Either use a movie clip created during authoring or use one created programmatically. To start with a fresh canvas, use *MovieClip.createEmptyMovieClip( )* . You can draw on the main timeline by specifying `_root` as the movie clip in which to draw, but drawing into a clip attached to the main timeline is the preferred approach.

The following code creates an empty movie clip, `drawing_mc` , and draws a red triangle in it, 100 pixels below and to right of the clip's registration point:

```
// Create the clip
this.createEmptyMovieClip("drawing_mc", 1);
// Position the drawing pen
var startX = 100;
var startY = 100;
drawing_mc.moveTo(startX , startY);
// Start the shape
drawing_mc.beginFill(0xFF0000, 100);
// Draw the lines of the triangle
drawing_mc.lineTo(startX+100, startY );    // Right 100 pixels
drawing_mc.lineTo(startX+50, startY+100);  // Left 50 pixels, down 100 pixel
drawing_mc.lineTo(startX, startY);         // Return to start point to close
// End the shape
drawing_mc.endFill();
```

The stroke defined by the most recent call to *lineStyle ( )* is automatically added to the shape. If *lineStyle ( )* has never bee the shape is not stroked. For example, the following code draws the same red triangle, but with a 1-pixel black line aroun perimeter. This time we specify positions using numeric literals only, rather than the variables `startX` and `startY` :

```
this.createEmptyMovieClip("drawing_mc", 1);
// Indicate a 1-pixel black stroke for the shape
drawing_mc.lineStyle(1, 0x000000);
drawing_mc.moveTo(100, 100);
drawing_mc.beginFill(0xFF0000, 100);
drawing_mc.lineTo(200, 100);
drawing_mc.lineTo(150, 200);
drawing_mc.lineTo(100, 100);
drawing_mc.endFill();
```

Even though Flash attempts to fix improperly specified shapes (e.g., a shape with no *endFill ( )* method), predictable resul guaranteed only when these shape-drawing rules are observed:

- The line around the shape's perimeter must have identical start and end points.
- The shape must be formally closed with a call to *endFill( )* .
- The *beginFill( )* call must be preceded immediately by a *moveTo( )* call that defines the starting point of the shape t filled.

For example, the following code draws a line to the point (100, 100) and then draws our red triangle starting at that point. the triangle is drawn, *moveTo ( )* sets the starting point to (100, 100), even though the pen need not move. If the call to *m)* is omitted or placed after the *beginFill ( )* call, Flash will not properly render the line and triangle.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

```
this.createEmptyMovieClip("drawing_mc", 1);
drawing_mc.lineStyle(1);
drawing_mc.lineTo(100, 100);  // Draw the line
// Now draw a triangle
drawing_mc.moveTo(100, 100);  // Set the triangle's starting point. Required
drawing_mc.beginFill(0xFF0000);
drawing_mc.lineTo(200, 100);
drawing_mc.lineTo(150, 200);
drawing_mc.lineTo(100, 100);
drawing_mc.endFill();
```

Movie clips that contain content drawn via ActionScript behave exactly like movie clips with content created in the Flash authoring tool. For example, the `_width` and `_height` properties reflect content drawn at runtime, as do calls to *getB* and *hitTest ( )* . Furthermore, provided they're not completely transparent, shapes created with calls to *beginFill ( )* or *beginGradientFill ( )* are available as hit test areas for *onRollover ( )* and *onPress ( )* events or for use as a mask (see *MovieClip.setMask( )*).

To draw a fully contained hole within a shape, draw the shape and then the hole within a single *beginFill ( ) /endFill ( )* bl example, the following code draws a "square doughnut":

```
this.lineStyle(3);
this.beginFill(0xFF0000, 100);
  // First square
  this.moveTo(100, 100);
  this.lineTo(200, 100);
  this.lineTo(200, 200);
  this.lineTo(100, 200);
  this.lineTo(100, 100);

  // Second square (the hole in the doughnut!)
  this.moveTo(150, 150);
  this.lineTo(170, 150);
  this.lineTo(170, 170);
  this.lineTo(150, 170);
  this.lineTo(150, 150);
this.endFill;
```

Note that bitmap fills are not directly supported by the Drawing API, but it is possible to achieve a similar effect by loadir image into a movie clip, then drawing a shape in a second clip and applying the second clip as a mask to the image clip. T clip must be fully loaded before the mask is applied. For example:

```
// Create the mask clip
this.createEmptyMovieClip("mask_mc", 2 );

// Draw the mask
mask_mc.beginFill(0xFF0000);
mask_mc.moveTo(100,100);
mask_mc.lineTo(200,100);
mask_mc.lineTo(200,200);
mask_mc.lineTo(100,200);
mask_mc.lineTo(100,100);
mask_mc.endFill();
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

```
// Create a clip to hold the image to be masked
this.createEmptyMovieClip("image_mc", 1);
// Load image to be masked
image_mc.loadMovie("image.jpg");

// Apply the mask once the image is loaded
this.onEnterFrame = function () {
  // Wait until the image is loaded
  if (image_mc._width > 0 && image_mc._totalframes =  = image_mc._framesload
    // The image has loaded, so apply the mask
    this.setMask(mask_mc);
    // Stop checking for the image to load
    delete this.onEnterFrame;
  }
}
```

Example

The Drawing API does not include methods for drawing primitive shapes, such as rectangles, circles, or polygons. These ı custom-created by ActionScript developers, as shown in Example 18-2, which adds *drawRect ( )* and *drawCircle ( )* meth movie clips.

Example 18-2. Custom drawRect( ) and drawCircle( ) methods

```
// Method: MovieClip.drawRect()
//   Desc: Draws a rectangle with optional fill.
// Params: x           Horizontal position of rectangle's top-left corner.
//         y           Vertical position of rectangle's top-left corner.
//         w           Width of rectangle, in pixels.
//         h           Height of rectangle, in pixels.
//         RGB         Color of rectangle.
//         alpha       Transparency of rectangle.
MovieClip.prototype.drawRect = function (x, y, w, h, RGB, alpha) {
  this.moveTo(x, y);
  this.beginFill(RGB, alpha);
  this.lineTo(x+w, y);
  this.lineTo(x+w, y+h);
  this.lineTo(x, y+h);
  this.lineTo(x, y);
  this.endFill;
};

// drawRect() usage:
// Set line width to 3 pixels.
this.lineStyle(3);
// Draw two overlapping rectangles with a transparency effect. Ooohhh...
this.drawRect(300, 200, 50, 20, 0x0000FF, 100);
this.drawRect(320, 210, 100, 120, 0x00FF00, 60);

// Method: MovieClip.drawCircle()
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

```
//   Desc: Draws a circle outline (no fill) as a series of n curves.
// Params: x          Horizontal position of circle's center.
//         y          Vertical position of circle's center.
//         segments   Number of curves used to draw the circle (defaults to
MovieClip.prototype.drawCircle = function (x, y, r, segments) {
  // Variables to hold the end and control points of each curve.
  var endX, endY, controlX, controlY;
  // Use 8 segments if not specified.
  if (segments =  = undefined) {
    segments = 8;
  }
  // Calculate how many degrees are in each segment
  // (360/segments), and convert that to radians.
  var segmentAngle = (360/segments)/180 * Math.PI;
  // Calculate the distance to the control point of each curve.
  var controlDist = r / Math.cos(segmentAngle/2);
  // Start drawing on the circle's positive X-axis.
  this.moveTo(x+r, y);
  // Draw the circle segments.
  for (var i = 1; i <= segments; i++) {
    // Calculate the end point of this curve.
    endX = x + r * Math.cos(i*segmentAngle);
    endY = y - r * Math.sin(i*segmentAngle);

    // Calculate the control point of this curve.
    controlX = x + controlDist * Math.cos(i*segmentAngle-segmentAngle/2);
    controlY = y - controlDist * Math.sin(i*segmentAngle-segmentAngle/2);

    // Draw this curve.
    this.curveTo(controlX, controlY, endX, endY);
  }
};

// drawCircle() usage:
// Outlined circle
this.lineStyle(1);
this.drawCircle(200, 200, 100);
// Outlined and filled circle
this.lineStyle(3, 0xFF0000);
this.beginFill(0x0000FF);
this.drawCircle(300, 250, 50);
this.endFill();
```

Ric Ewing has posted more custom drawing methods at:
http://www.formequalsfunction.com/downloads/drawmethods.html

See Also

MovieClip.beginGradientFill( ) , *MovieClip.clear( )* , *MovieClip.curveTo( )* , *MovieClip.endFill( )* , *MovieClip.lineStyle(*

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

MovieClip.lineTo( ) , *MovieClip.moveTo( )* ; Chapter 13

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Chapter 18. ActionScript Language Reference

MovieClip.beginGradientFill( ) Method Flash 6

starts a gradient-filled shape

```
mc.beginFill(fillType, colors, alphas, ratios, matrix)
```

Arguments

fillType

A string indicating the type of gradient, either "linear" or "radial". If any other value is specified, no gradient is dra

colors

An array of integers specifying the colors in the gradient. Normally supplied in RGB hexadecimal triplet notation–`RRGGBB`, where `RR`, `GG`, and `BB` are two-digit hex numbers representing Red, Green, and Blue. For exampl hex integer `0xFF0000` indicates red. For a complete discussion of RGB triplet notation, see the *Color* class.

alphas

An array of integers between 0 and 100, specifying the percentage opacity (or, conversely, the transparency) of the colors in the gradient. 0 is completely transparent, whereas 100 is completely opaque. Values above 100 are interpı as 100; those below 0 are interpreted as 0. If the number of elements in the `alphas` array does not match the nu of elements in the `colors` array, no gradient is drawn.

ratios

An array of integers specifying the spatial distribution of colors in the gradient on a scale from 0 to 255. Each elem in the array indicates the relative position where the corresponding color from the `colors` array will display as a unblended color. If the number of elements in the `ratios` array does not match the number of elements in the `colors` array, no gradient is drawn.

matrix

An object specifying the position, size, rotation, and skew of the gradient. The object takes one of two forms: a sim bounding box, where the gradient's position, size, and rotation are supplied through intuitive properties, or a raw 3 transformation matrix that maps a unit gradient centered about (0,0) to a final size, rotation, skew, and position. Se details under the Description.

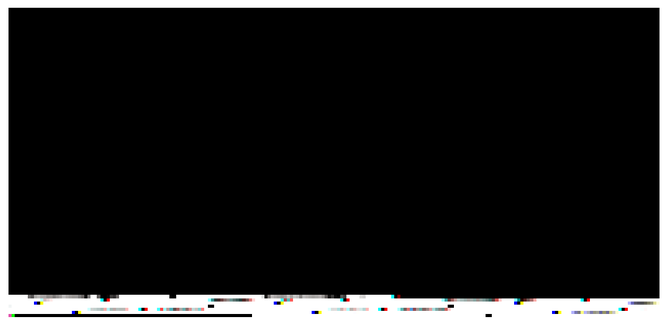
Description

Like *beginFill ( )*, *beginGradientFill ( )* starts a shape, but it fills that shape with a linear or radial gradient (shown in Figu 18-1) instead of a solid fill.

Figure 18-1. A radial and a linear gradient

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir



The gradient is defined, positioned, sized, and rotated by the `fillType` , `colors` , `alphas` , `ratios` , and `ma` parameters. The filled shape appears *above* all Drawing API content already in `mc` , but *beneath* all other content in `mc` movie clips, shapes, text, and buttons placed at authoring time, or content attached at runtime via *attachMovie( )* , *duplicateMovieClip( )* , or *createTextField( )*).

Just as with *beginFill ( )* we start drawing a gradient-filled shape with a call to *moveTo ( )* , which indicates the starting pc of the outline we'll draw around the shape. Next, we call *beginGradientFill ( )* to start the shape and define the attributes o gradient. We then draw the shape's outline with *lineTo ( )* and/or *curveTo ( )* calls. Conceptually, the gradient is distinct f the shape, as though the two were separated onto two layers, with the gradient beneath the shape. The shape acts as a hole through which the gradient layer is visible (that is, the gradient is "clipped" by the shape). For example, if we draw a 100 x 100 square at position (25,0) and a 75 x 300 two-color linear gradient at position (0,0), the blended portion of the gradient extends 50 pixels into the square, and the remainder of the square is filled with the rightmost color in the gradient, as shov Figure 18-2.

Figure 18-2. A linear gradient and a square

To make the gradient span the entire square perfectly, we'd have to change its size to 100 x 100, and position it at (25,0), t top-left corner of the square. Let's see how this works in code, first with the offset gradient, then with the perfectly alignec First, we make a clip in which to draw the square:

```
this.createEmptyMovieClip("drawing_mc", 1);
```

Then we position the drawing pen at (25,0):

```
drawing_mc.moveTo(25, 0);
```

Now we have to define the gradient. We'll store each of *beginGradientFill( )* 's arguments in a variable for easy reading. T

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

fillType parameter should be "linear" or "radial", indicating whether the bands of color in the gradient will form stra lines or circles. We use a "linear" gradient (straight lines):

```
var fillType = "linear";
```

Next, we specify an array of color values (the *colors* parameter); these are the bands of color that will be blended toge in our gradient. We have two colors, white and black, but you can specify as many as you like:

```
var colors = [0xFFFFFF, 0x000000];
```

Next, we specify the transparency of each color as an array of integers between 0 and 100 (the *alphas* parameter). Ea element in *alphas* corresponds to an element in *colors* (both arrays must have the same number of elements). An a of 0 is completely transparent, whereas 100 is completely opaque. Our colors are not transparent, so we set them both to 1

```
var alphas = [100, 100];
```

Next, we define the proportional position of each color band in our gradient as an array of integers between 0 and 255 (the *ratios* parameter). As before, the *ratios* array must have the same number of elements as the *colors* array. A of 0 means the color is pure at the leftmost (linear) or innermost (radial) point of the gradient. A value of 255 means the c is pure at the rightmost (linear) or outermost (radial) point of the gradient. We want our gradient to blend evenly from whi to black across its entire width, so we set the position of pure white to 0, and the position of pure black to 255:

```
var ratios = [0, 255];
```

Finally, we set the position, width, height, and rotation of our gradient as an object (the *matrix* parameter). We set the *matrix* object's `matrixType` property to "box" and then provide values for the following properties: `x` (horizontal position), `y` (vertical position), `w` (width), `h` (height), and `r` (rotation). The `x` and `y` properties specify the gradient's top-le corner, relative to the clip's registration point, and the `r` property specifies the gradient's angle of rotation, in radians, not degrees. Our code calls a custom *toRadians ( )* function that converts degrees to radians:

```
var matrix = { matrixType:"box", x:0, y:0, w:75, h:300, r:toRadians(0) };
function toRadians (deg) {
  return deg * Math.PI/180;
}
```

With our gradient arguments defined, we can now invoke *beginGradientFill ( )* :

```
drawing_mc.beginGradientFill(fillType, colors, alphas, ratios, matrix);
```

And then we simply draw our square, exactly as we would with a solid fill, being sure to return to the starting point (25,0) with our last call to *lineTo ( )* :

```
drawing_mc.lineTo(125, 0);
drawing_mc.lineTo(125, 100);
drawing_mc.lineTo(25, 100);
drawing_mc.lineTo(25, 0);
```

Finally, we formally close the shape with *endFill ( )* , which completes our gradient-filled square:

```
drawing_mc.endFill();
```

Now, how about aligning our gradient perfectly with the square? No problem, we just change the gradient's starting positi the shape's origin, (25,0), and set its size to 100 x 100 by adjusting our *matrix* object:

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

```
var matrix = { matrixType:"box", x:25, y:0, w:100, h:100, r:toRadians(0) };
```

By setting the `r` property of *`matrix`* to the number of radians in 45 degrees, using *toRadians(45)* , we can create a diag gradient instead:

```
var matrix = { matrixType:"box", x:25, y:0, w:100, h:100, r:toRadians(45) };
```

To fill the nicely aligned box with a white circular gradient, we can set *`fillType`* to radial:

```
var fillType = "radial";
```

To switch the order of the colors, we simply reverse the *`colors`* array, as follows:

```
var colors = [0x000000, 0xFFFFFF];
```

The stroke defined by the most recent call to *lineStyle ( )* is automatically added to the shape. If *lineStyle ( )* has never bee called, the shape is not stroked. For example, placing the following code before our *moveTo ( )* statement adds a red strok our square:

```
drawing_mc.lineStyle(1, 0xFF0000);
```

For most purposes, the bounding box object we used for *`matrix`* in our square example provides ample gradient contro However, a gradient's position, rotation, scale, and skew can also be specified by a 3 x 3 transformation matrix applied to unit gradient centered at position (0,0). Transformation matrices are useful when programmatically generating fills, as is necessary in, say, a 3D engine. To use a 3 x 3 transformation matrix, we set the *`matrix`* parameter of *beginGradientFi* to an object with the properties `a` , `b` , `c` , `d` , `e` , `f` , `g` , `h` , and `i` , corresponding to the following positions in our 3 x 3 matrix:

```
---------------
|  a   b   c  |
|  d   e   f  |
|  g   h   i  |
---------------
```

To scale, rotate, and translate (move) the unit gradient, we use the following transformation matrices:

```
     SCALE                TRANSLATE                ROTATE (clockwise)
---------------       ---------------       -------------------------------
|  sx  0   0  |       |  1   0   0  |       | cos(theta)  sin(theta)
|  0   sy  0  |       |  0   1   0  |       | -sin(theta) cos(theta)
|  0   0   1  |       |  tx  ty  1  |       | 0           0
---------------       ---------------       -------------------------------
sx = x scale factor     tx = amount of x-shift
sy = y scale factor     ty = amount of y-shift
```

In the simplest case, where we neither scale, rotate, nor translate the gradient, the matrices reduce to:

```
     SCALE                TRANSLATE                ROTATE (clockwise)
---------------       ---------------       -------------------------------
|  1   0   0  |       |  1   0   0  |       | 1           0
|  0   1   0  |       |  0   1   0  |       | 0           1
|  0   0   1  |       |  0   0   1  |       | 0           0
---------------       ---------------       -------------------------------
sx = x-scale factor     tx = amount of x-shift
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

```
sy = y-scale factor     ty = amount of y-shift
```

and our matrix object is set as follows:

```
var matrix = { a:1, b:0, c:0, d:0, e:1, f:0, g:0, h:0, i:1 };
```

To scale the unit gradient to 100 x 100, we set the `a` and `e` properties to 100, leaving the `i` property as 1 and the others a as follows:

```
var matrix = { a:100, b:0, c:0, d:0, e:100, f:0, g:0, h:0, i:1 };
```

This transformation is close to the one we specified earlier for our square's gradient, but it is not identical. The x-scale and scale factors of the matrix (`a` and `e`) cause the unit gradient, which is centered around the origin (0,0), to be scaled up to a height and width of 100. Scaling alone, however, does not change the center point of the gradient—it remains at (0,0) whe the translation properties (`g` and `h`) are set to 0.

To also move the gradient's origin to position (25,0), we set the horizontal translation parameter (`g`) to move the gradien pixels to the right, as follows:

```
var matrix = { a:100, b:0, c:0, d:0, e:100, f:0, g:25, h:0 , i:1 };
```

This produces a gradient that is centered about the point (25,0).

However, our earlier matrix positioned the gradient's top-left corner, not its center, at (25,0). To center the gradient in our square (as we did earlier), we need to translate from the point (25,0) to the middle of the destination by adding 50 to the horizontal and vertical coordinates (50 is half of our square's width and height).

Therefore, to center our gradient, we set `g` to 75 (which is 25 + 50) and `h` to 50 (which is 0 + 50), as follows:

```
var matrix = { a:100 , b:0 , c:0 , d:0 , e:100 , f:0 , g:75 ,h:50 , i:1 };
```

For comparison, here's the complete square code, showing both ways of specifying our *beginGradientFill( )* 's *`matrix`* argument:

```
// Convenient degree-to-radian converter.
function toRadians (deg) {
  return deg * Math.PI/180;
}

// Define our beginGradientFill() parameters.
var fillType = "linear";
var colors = [0xFFFFFF, 0x000000];
var alphas = [100, 100];
var ratios = [0, 255];
// Use either the box matrix version:
var matrix = { matrixType:"box", x:25, y:0, w:100, h:100, r:toRadians(0) };
// or the transformation matrix version:
var matrix = { a:100 , b:0 , c:0 , d:0 , e:100 , f:0 , g:75 ,h:50 , i:1 };

// Make the movie clip.
this.createEmptyMovieClip("drawing_mc", 1);

// Draw the gradient-filled square.
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

```
drawing_mc.moveTo(25, 0);
drawing_mc.beginGradientFill(fillType, colors, alphas, ratios, matrix);
drawing_mc.lineTo(125, 0);
drawing_mc.lineTo(125, 100);
drawing_mc.lineTo(25, 100);
drawing_mc.lineTo(25, 0);
drawing_mc.endFill();
```

By combining matrices, we can use a single matrix to describe scale, rotation, and translation together. Matrix multiplicati outside the scope of this discussion; however, Macromedia provides a *TransformMatrix* class on the Flash MX CD under: *Goodies\Macromedia\Other Samples\transformmatrix.as* . It defines easy-to-use methods for specifying and combining transformation matrices.

Usage

Even though Flash attempts to fix improperly specified shapes (e.g., a shape with no *endFill ( )* method), predictable resul be guaranteed only when the shape-drawing rules described under *MovieClip.beginFill( )* are observed.

Example

The following code draws a triangle with a red, green, and blue radial gradient. The triangle is centered about the clip's registration point, and the gradient is centered within the triangle:

```
// Create an empty clip for drawing, and set the pen position
this.createEmptyMovieClip("drawing_mc", 1);
drawing_mc._x = drawing_mc._y = 200;
drawing_mc.lineStyle(1, 0xFF0000);
drawing_mc.moveTo(0, -50);

// Set gradient parameters
var fillType = "radial";
var colors = [0xFF0000, 0x00FF00, 0x0000FF];
var alphas = [100, 100, 100];
var ratios = [0, 127, 255];
var matrix = { matrixType:"box", x:-50, y:-35, w:100, h:100, r:toRadians(0)
function toRadians (deg) {
  return deg * Math.PI/180;
}

// Start the shape
drawing_mc.beginGradientFill(fillType, colors, alphas, ratios, matrix);
// Draw the lines of the triangle
drawing_mc.lineTo(50, 50);
drawing_mc.lineTo(-50, 50);
drawing_mc.lineTo(0, -50);
// End the shape
drawing_mc.endFill();
```

See Also

MovieClip.beginFill( ) , *MovieClip.clear( )* , *MovieClip.curveTo( )* , *MovieClip.endFill( )* , *MovieClip.lineStyle( )* ,

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

MovieClip.lineTo( ) , *MovieClip.moveTo( )* ; Chapter 13

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Chapter 18. ActionScript Language Reference

MovieClip.clear( ) Method — Flash 6

erases all Drawing API content from a movie clip

```
mc.clear()
```

Description

The *clear( )* method erases all lines (strokes) and shapes (fills) created with the *MovieClip* Drawing API methods in `mc`, but it does not affect content created in the Flash authoring tool. It also clears the current line style (as if *lineStyle(undefined)* were called) and returns the drawing pen to position (0,0). To activate lines for subsequent drawings, call *lineStyle( )* before invoking any drawing methods.

You can't erase author-time content using the Drawing API. Instead, use the `_visible` property to hide a movie clip.

Example

The following code draws a single line with a random line style on every frame. It uses *clear( )* to erase the line drawn in the previous frame:

```
// A custom function to return a random integer
Math.randomInt = function (minVal, maxVal) {
  r = Math.random();
  return minVal + Math.floor(r * (maxVal + 1 - minVal));
}

this.createEmptyMovieClip("drawing_mc", 1);
drawing_mc.onEnterFrame = function () {
  this.clear();
  this.lineStyle(Math.randomInt(1, 40), Math.randomInt(1, 0xFFFFFF));
  this.moveTo(Math.randomInt(1, 550),Math.randomInt(1, 400));
  this.lineTo(Math.randomInt(1, 550),Math.randomInt(1, 400));
}
```

Reader Exercise 1: Try removing the call to *clear( )* to see how calls to the drawing functions are cumulative until *clear( )* is invoked.

Reader Exercise 2: Using *clear( )* erases the Drawing API content from the specified clip only. Write a custom function to erase all Drawing API content from all movie clips, starting with the specified clip and recursively erasing the dynamic content from any nested clips. Use a *for-in* loop to detect nested clips. You can use your function with the main timeline to erase all dynamic content.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

See Also

MovieClip.beginFill( ), *MovieClip.beginGradientFill( )*, *MovieClip.curveTo( )*, *MovieClip.endFill( )*, *MovieClip.lineStyle( )*, *MovieClip.lineTo( )*, *MovieClip.moveTo( )*, `MovieClip._visible`; Chapter 13

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Chapter 18. ActionScript Language Reference

MovieClip.createEmptyMovieClip( ) Method Flash 6

creates a new, empty MovieClip instance

```
mc.createEmptyMovieClip(instanceName, depth)
```

Arguments

instanceName

A string specifying the instance name of the clip to create. The name must adhere to the rules for creating an identifier outlined under Section 15.5. To enable code hinting in the Flash authoring tool, use clip instance names that include the suffix "`_mc`", as in `ball_mc`.

depth

An integer between -16384 and 1048575 (inclusive), specifying the level on which to place the new clip inside *mc*. Higher depths obscure lower depths, so a depth of 6 is in front of 5, which is in front of 4, and so on. Depths above -1 are reserved for dynamically generated content. Therefore, in most cases, you should use a depth between 0 and 1048575 with *createEmptyMovieClip( )*. However, when creating dynamic content that you want to appear beneath all author-time content in *mc* (e.g., a run-time background), use depth -16384. Depths from -16383 to -1 (inclusive) are reserved for author-time content and shouldn't be used with *createEmptyMovieClip( )*. For important details, see Section 13.4.

Returns

A reference to the new *MovieClip* instance.

Description

The *createEmptyMovieClip( )* method creates a new, blank movie clip instance and attaches it to *mc* at the specified *depth*. If *mc* is the main movie, the new instance is placed in the top-left corner of the Stage. If *mc* is a movie clip instance, the new instance is placed at *mc*'s registration point. The new instance is placed in *mc*'s content stack at the specified *depth*, replacing any previous occupant of that depth.

The *createEmptyMovieClip( )* method is used to create content at runtime without requiring any Library symbol. For example, to create a spaceship dynamically, we can create an empty clip and then use the *MovieClip* Drawing API methods to draw the ship. Likewise, to create a form or interface panel at runtime, we can create an empty clip and then populate it with text fields, as follows:

```
this.createEmptyMovieClip("form_mc", 1);
form_mc.createTextField("firstName_txt", 1, 0, 0, 200, 20);
form_mc.firstName_txt.border = true;
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

```
form_mc.firstName_txt.type = "input";
form_mc.createTextField("lastName_txt", 2, 0, 30, 200, 20);
form_mc.lastName_txt.border = true;
form_mc.lastName_txt.type = "input";
form_mc.createTextField("email_txt", 3, 0, 60, 200, 20);
form_mc.email_txt.border = true;
form_mc.email_txt.input = "input";
```

An empty movie clip can also be used simply as a holder for events, such as *onEnterFrame( )*, *onKeyDown( )*, *onMouseDown( )*, etc:

```
this.createEmptyMovieClip("process_mc", 1);
process_mc.onEnterFrame = function () {
  // perform per-frame tasks here
}
```

The *createEmptyMovieClip( )* method returns a reference to the clip it creates. We can store this reference in a variable for easy access. For example, the following code creates an empty clip deeply nested inside an application's interface but stores a reference to the clip in the variable `fileMenu`:

```
// Create the clip.
var fileMenu = appPanels.editPanel.menuBar.createEmptyMovieClip
       ("fileMenu_mc", 1);
// Now manipulate the clip through the variable fileMenu.
fileMenu._x = 20;
fileMenu._y = 50;
```

A movie clip created with *createEmptyMovieClip( )* can be used as a mask for other clips (see *MovieClip.setMask( )*) and can receive button events such as *onPress( )* and *onRelease( )*. However, strokes and transparent fills are not recognized for masking or hit-testing for button events.

To delete a movie clip, use *MovieClip.removeMovieClip( )*. However, note that a movie clip placed on a negative depth cannot be removed via *removeMovieClip( )*. To remove a clip at a negative depth, first swap it to a positive depth, then remove it:

```
theClip_mc.swapDepths(1234);   // Pick a positive depth to swap to
theClip_mc.removeMovieClip();  // Remove the clip
```

See the *removeMovieClip( )* global function Description for more important details.

Usage

Depths above 1048575 are partially functional but not supported. For example, though a movie clip can be attached to depth 1048576, it cannot be removed. An attempt to attach a movie clip to a depth below -16384 fails silently.

See Also

duplicateMovieClip( ), MovieClip.attachMovie( ), MovieClip.createTextField( ), MovieClip.duplicateMovieClip( ), MovieClip.getDepth( ), MovieClip.removeMovieClip( ), MovieClip.swapDepths( )

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Chapter 18. ActionScript Language Reference

MovieClip.createTextField( ) Method Flash 6

creates a new TextField object

```
mc.createTextField(instanceName, depth, x, y, width, height)
```

Arguments

instanceName

A string specifying the instance name of the *TextField* object to create. The name must adhere to the rules for creating an identifier outlined under Section 15.5. To enable code hinting in the Flash authoring tool, use text field instance names that include the suffix "`_txt` ", as in `firstName_txt` .

depth

An integer, between -16384 and 1048575 (inclusive), specifying the level on which to place the new text field inside `mc` . Higher depths obscure lower depths, so a depth of 6 is in front of 5, which is in front of 4, and so on. Depths above -1 are reserved for dynamically generated content. Therefore, in most cases, you should use a depth between 0 and 1048575 with *createTextField( )* . However, when creating dynamic content that you want to appea beneath all author-time content in `mc` (e.g., a run-time background), use depth -16384. Depths from -16383 to -1 (inclusive) are reserved for author-time content and shouldn't be used with *createTextField( )* . For important details, see Section 13.4.

x

A floating-point number specifying the horizontal position of the newly created field's left edge, in pixels, relative to the coordinate space of `mc` . See `TextField ._x` .

y

A floating-point number specifying the vertical position of the newly created field's top edge, in pixels, relative to the coordinate space of `mc` . See `TextField ._y` .

width

A floating-point number specifying the width of the newly created field's bounding box, in pixels. See `TextField ._width` .

height

A floating-point number specifying the height of the newly created field's bounding box, in pixels. See `TextField ._height` .

Description

The *createTextField ( )* method creates a new on-screen text field controlled by the ActionScript *TextField* object named

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

instanceName . It is the runtime equivalent of creating a text field with the Text tool in the Flash authoring tool. If *mc* is the main movie, the new text field is placed at the point (*x* , *y*), relative to the top-left corner of the Stage. If *mc* is a movie clip instance, the new text field is placed at the point (*x* , *y*), relative to *mc* 's registration point. The new text field is placed in *mc* 's content stack at the specified *depth* , replacing any previous occupant of that depth. The text field's default `type` is "dynamic".

Once the text field has been created, it is controlled by the methods and properties of the *TextField* class. For instructions on deleting a text field, see the *TextField.removeTextField( )* method.

Example

The following code creates a text field on the main timeline of a movie (`_root`), at depth 2, positioned at point (0, 0) (the top-left corner of the movie), with a bounding box width of 200 and height of 20. Once the field is created, the text "Hello world!" is displayed in it:

```
_root.createTextField("welcome_txt", 1, 0, 0, 200, 20);
_root.welcome_txt.text = "Hello world!";
```

The following code creates the same text field, but also gives it a border and positions it at the center of the movie's Stage:

```
// Create the text field.
_root.createTextField("welcome_txt", 1, 0, 0, 200, 20);
// Add a border and some text.
_root.welcome_txt.border = true;
_root.welcome_txt.text = "Welcome to my site!";
// Make the border match the text size.
_root.welcome_txt.autoSize = true;

// Set the alignment mode of the movie.
Stage.align = "LT";
Stage.scaleMode = "noScale";

// Center the text field any time the movie changes size.
_root.welcome_txt.onResize = function () {
  this._x = (Stage.width/2) - (this._width/2);
  this._y = (Stage.height/2) - (this._height/2);
}
// Make the text field listen for the Stage.onResize() event.
Stage.addListener(_root.welcome_txt);

// Center the text field manually. This is required in the Standalone Player
// which does not automatically trigger onResize() when the movie loads.
_root.welcome_txt.onResize();
```

See Also

MovieClip.createEmptyMovieClip( ) , *TextField.getDepth( )* , *TextField.removeTextField( )* ; the *Stage* class, the *TextField* class

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Chapter 18. ActionScript Language Reference

MovieClip._currentframe Property — Flash 4

the frame number of the playhead of a clip or movie — ***read-only***

mc._currentframe

Description

The integer _currentframe property represents the frame number at which the playhead of *mc* currently resides. Note that the first frame is 1, not 0; therefore, _currentframe ranges from 1 to *mc*._totalframes.

Example

```
// Send the playhead backward two frames from its current location
this.gotoAndStop(this._currentframe - 2);
```

See Also

MovieClip.gotoAndPlay( ), *MovieClip.gotoAndStop( )*, *MovieClip*._totalframes

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

MovieClip.curveTo( ) Method Flash 6

draws a curved line

```
mc.curveTo(controlX, controlY, anchorX, anchorY)
```

Arguments

controlX

The horizontal position of the off-curve control point of the quadratic Bezier curve, as a floating-point number. Measured relative to `mc`'s registration point.

controlY

The vertical position of the off-curve control point of the quadratic Bezier curve, as a floating-point number. Measured relative to `mc`'s registration point.

anchorX

The horizontal end position of the curve to be drawn, as a floating-point number. Measured relative to `mc`'s registration point.

anchorY

The vertical end position of the curve to be drawn, as a floating-point number. Measured relative to `mc`'s registration point.

Description

The *curveTo( )* method draws a quadratic Bezier curve from the current drawing pen position to the point (`anchorX`, `anchorY`) using an off-curve control point of (`controlX`, `controlY`). The tangents to the curve at the start and end points both pass through the control point. Conceptually speaking, the straight line that would run from the pen position to the end point (`anchorX`, `anchorY`) is pulled by the control point (`controlX`, `controlY`) to make a curve. An alternate method that draws a curve through three specific points and does not require the user to supply a control point is shown in the following Example.

The stroke characteristics of the curve (thickness, color, alpha) are determined by the most recent call to *lineStyle( )*.

> If *lineStyle( )* hasn't been called, the stroke style will be `undefined`. In that case, *curveTo( )* moves the drawing pen position to (`anchorX`, `anchorY`), exactly as if *moveTo( )* had been called, but does not draw a curve on screen.

The curve, when drawn, appears *above* all Drawing API content already in `mc`, but *beneath* all other content in

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

mc (e.g., movie clips, shapes, text, and buttons placed at authoring time, or content attached at runtime via *attachMovie( )*, *duplicateMovieClip( )*, or *createTextField( )*).

The following code creates a movie clip named `drawing_mc` and draws a 4-point black curve from the drawing pen's default position (0,0) to the anchor point (100,0) using the control point (50,-100). The resulting curve is shown in Figure 18-3.

```
// Create a clip to draw in
this.createEmptyMovieClip("drawing_mc", 1);

// Set the stroke to 4-point, black
drawing_mc.lineStyle(4);

// Draw the curve
drawing_mc.curveTo(50, -100, 100, 0);

// Center the drawing clip on the Stage
drawing_mc._x = 200;
drawing_mc._y = 225;
```

Figure 18-3. A sample quatratic Bezier

After a curve is drawn, the drawing pen remains at the end point of the curve. Here, multiple calls to *curveTo( )* and/or *lineTo( )* can be used to draw complex curves or closed shapes, such as circles and polygons. For details, see Example 18-2 under *MovieClip.beginFill( )*.

If any of *curveTo( )*'s arguments are missing, the operation fails silently and the position of the drawing pen remains unchanged. For a description of the drawing pen, see the Discussion under *MovieClip.moveTo( )*.

For a primer on Bezier curve mathematics and programming, see:

http://www.ipm.sci-nnov.ru/~demidov/VRML/Splines/Intro/Bezier.htm

Example

Sometimes it is more convenient to specify three points on a curve rather than two anchor points and a control point. The following code adds a custom *curveThrough3Pts( )* method to all movie clips; the method accepts three points as arguments and draws a quadratic curve that passes through them. The second point is assumed to be halfway along the curve, in time (t = .5):

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

```
// Adapted from Robert Penner's drawCurve3Pts() method
MovieClip.prototype.curveThrough3Pts =
    function (startX, startY, throughX, throughY, endX, endY) {
  var controlX = (2 * throughX) - .5 * (startX + endX);
  var controlY = (2 * throughY) - .5 * (startY + endY);
  this.moveTo(startX, startY);
  this.curveTo(controlX, controlY, endX, endY);
};

// Usage
this.createEmptyMovieClip("drawing_mc", 1);
drawing_mc.lineStyle(1, 0xFF0000);
drawing_mc.curveThrough3Pts(100, 100, 150, 50, 200, 100);
```

See Also

MovieClip.beginFill( ), *MovieClip.beginGradientFill( )*, *MovieClip.clear( )*, *MovieClip.curveTo( )*, *MovieClip.endFill( )*, *MovieClip.lineStyle( )*, *MovieClip.lineTo( )*, *MovieClip.moveTo( )*

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Chapter 18. ActionScript Language Reference

MovieClip._droptarget Property — Flash 4

the path to the clip or movie on which a dragged clip was dropped — ***read-only***

`mc._droptarget`

Description

If `mc` is being dragged, `_droptarget` returns a string indicating the path to the clip over which `mc` hovers (if any). If `mc` is not hovering over a clip, `_droptarget` returns `undefined`. If `mc` was previously dragged and then dropped on a clip, `_droptarget` returns a string indicating the path to the clip upon which `mc` was dropped. The path is provided in slash notation. A movie clip is considered to be "over" another clip if the registration point of the dragged clip overlaps any portion of the target clip. If `mc` hovers over two or more overlapping clips, only the topmost clip is considered the `_droptarget`. To detect intersection between any number of overlapping clips, use *MovieClip.hitTest ( )*.

Example

The `_droptarget` property is convenient for creating drag-and-drop interfaces. The following example demonstrates how to create a simple shopping-basket interface using `_droptarget`. When the `item_mc` clip is dropped onto the `basket` clip, it is allowed to stay in the basket; otherwise, the `item_mc` is returned to its original location:

```
// Set original location
item_mc.origX = _x;
item_mc.origY = _y;

// Define a function to call when the item is clicked
item_mc.drag = function () {
  this.startDrag();
}

// Define a function to call when the item is dropped
item_mc.drop = function () {
  stopDrag();
  if (this._droptarget != "/basket") {
    this._x = this.origX;
    this._y = this.origY;
  }
}
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

```
// Assign our drag/drop callback to the appropriate item_mc button handlers
item_mc.onPress = item_mc.drag;
item_mc.onRelease = item_mc.onReleaseOutside = item_mc.drop;
```

Note that `_droptarget` stores a string in slash notation, not as a clip reference. To convert a `_droptarget` string to a movie clip reference, use the technique shown in the Example under the *eval ( )* global function.

See Also

MovieClip.startDrag( ) , *MovieClip.stopDrag( )*

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

MovieClip.duplicateMovieClip( ) Method Flash 6

create a copy of a movie clipFlash 5; initObject and return value added in

```
mc.duplicateMovieClip(newName, depth)
mc.duplicateMovieClip(newName, depth, initObject)
```

Arguments

newName

A string that will become the instance name of the duplicated clip. The name must adhere to the rules for creating an identifier outlined under Section 15.5.

depth

An integer between -16384 and 1048575 (inclusive), specifying the level on which to place the new clip inside `mc`'s parent. Higher depths obscure lower depths, so a depth of 6 is in front of 5, which is in front of 4, and so on. Depths above -1 are reserved for dynamically generated content. Therefore, in most cases, you should use a depth between 0 and 1048575 with *duplicateMovieClip( )*. However, when creating dynamic content that you want to appear beneath all author-time content in `mc` (e.g., a runtime background), use depth -16384. Depths from -16383 to -1 (inclusive) are reserved for author-time content and shouldn't be used with *duplicateMovieClip( )*. For important details, see Section 13.4.

initObject

An optional object whose properties are copied to the new movie clip instance. Not supported in Flash 5.

Returns

In Flash 6, *duplicateMovieClip( )* returns an object reference to the attached movie clip (in Flash 5, it always returns `undefined`).

Description

The *MovieClip.duplicateMovieClip( )* method is an alternative to the global *duplicateMovieClip( )* function. When invoked as a *MovieClip* method, *duplicateMovieClip( )* takes no `target` parameter (as it does with the global version)—it duplicates `mc` and returns a reference to the new duplicate. The *MovieClip* method syntax is less prone to user error than its global function counterpart because no `target` parameter is required.

For general usage instructions, see the global *duplicateMovieClip( )* function. For details on the `initObject` parameter, see *MovieClip.attachMovie( )* and Chapter 14.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Usage

The global form of *duplicateMovieClip( )* does not include support for the `initObject` parameter, nor does it return a reference to the newly duplicated clip (these limitations also apply to the movie clip method form in Flash 5).

Example

Make a duplicate of `ball1_mc` named `ball2_mc`. The duplicate is placed on depth 1 inside `ball1_mc`'s parent:

```
ball1_mc.duplicateMovieClip("ball2_mc", 1);
```

See Also

duplicateMovieClip( ), MovieClip.attachMovie( )

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Chapter 18. ActionScript Language Reference

MovieClip.enabled Property Flash 6

Boolean; allows or disallows button interaction ***read/write***

mc `enabled`

Description

Like *`Button`*`.enabled`, the Boolean *`MovieClip`*`.enabled` property determines whether *`mc`*'s button behavior is (`true`) or is not (`false`) enabled. By default, *`MovieClip`*`.enabled` is `true`; however, it has an effect only when one or more button events (*onPress( )*, *onRelease( )*, etc.) are defined for *`mc`*. When `enabled` is `false`, *`mc`* does not receive button event notifications, the hand cursor is not shown, and *`mc`*'s playhead does not jump automatically to the movie clip's `_up`, `_over` and `_down` frames in response to mouse interaction. However, *`mc`* will still play normally, and it can be focused with the Tab key. To disable tab focus, set *`mc`*`.tabEnabled` to `false`.

For information on implementing button behavior for a movie clip, see the *MovieClip* class Description, *`MovieClip`*`.hitArea`, and Section 13.9.

Usage

The Tab key does not work reliably in Test Movie mode unless Control Disable Keyboard Shortcuts is enabled. Be sure to test keyboard behavior in the Standalone Player, if applicable, and in the Flash Player for all targeted web browsers.

See Also

`Button``.enabled`, *`MovieClip`*`.hitArea`, *`MovieClip`*`.tabEnabled`, *`MovieClip`*`.useHandCursor`; Section 13.9

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Chapter 18. ActionScript Language Reference

MovieClip.endFill( ) Method Flash 6

terminates the fill begun with beginFill( ) or beginGradientFill( )

```
mc.endFill()
```

Description

The *endFill( )* method terminates the filling of a shape that was begun with an earlier call to *beginFill( )* or *beginGradientFill( )*, causing the shape to be drawn on-screen. If the current drawing pen position is not the same as when *beginFill( )* or *beginGradientFill( )* was called, Flash closes the shape with a straight line and applies the fill. However, creating open shapes can have unpredictable results and is not recommended.

For complete details on drawing shapes, see *MovieClip.beginFill( )* and *MovieClip.beginGradientFill( )*.

Example

```
// Draws a four-sided shape on the main timeline
_root.moveTo(225, 200);
_root.beginFill(0x00FF00, 50);
_root.lineTo(100,100);
_root.lineTo(0,100);
_root.lineTo(0,300);
_root.lineTo(225,200);
_root.endFill();
```

See Also

MovieClip.beginFill( ), MovieClip.beginGradientFill( ), MovieClip.clear( ), MovieClip.curveTo( ), MovieClip.lineStyle( ), MovieClip.lineTo( ), MovieClip.moveTo( )

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Chapter 18. ActionScript Language Reference

MovieClip.focusEnabled Property Flash 6

allow the clip to be focused programmatically ***read/write***

mc.focusEnabled

Description

When the Boolean `focusEnabled` property is `false` or `undefined` (the default) and *`mc`* defines no button handlers (*onPress ( )* , *onRelease ( )* , etc.), *`mc`* cannot be focused via *Selection.setFocus ( )* . When `focusEnabled` is `true` , *`mc`* can be focused via *Selection.setFocus ( )* at any time, even when no button handlers are defined. When *`mc`* defines at least one button handler, it can still be focused via *Selection.setFocus ( )* , even when `focusEnabled` is `false` . To disable focus for a movie clip with button handlers, remove the handlers and set `focusEnabled` to `false` . For example, the following code removes the *onPress ( )* handler of `theClip_mc` :

```
delete theClip_mc.onPress;
```

We use `focusEnabled` when creating movie clips that act as keyboard-controlled interface elements, such as a modal dialog box with Yes and No buttons activated by the keys "Y" and "N." The dialog box should respond to "Y" and "N" only when it has focus. The following code shows a simplified function that creates a dialog box clip and focuses it:

```
function displayAlert () {
  // appInterface is a movie clip containing the main application interface.
  // getFreeDepth(), not shown, is a function that returns an unoccupied
  // depth on which to place the dialog clip. dialog is the dialog box clip.
  var dialog = appInterface.attachMovie("dialog",
                                        "alertDialog",
                                        appInterface.getFreeDepth());
  // Allow the dialog box to be focused.
  dialog.focusEnabled = true;
  // Focus the dialog box.
  Selection.setFocus(dialog);
}
```

To highlight our `dialog` clip when it is focused, we can create frames with the special labels `_over` and `_up` in the `dialog` clip's timeline. The `_up` and `_over` frames will be displayed automatically by Flash when the clip loses and receives focus, respectively. You also might want to set the clip's `_focusrect` property to `false` . For a discussion of special focus issues for keyboard-controlled movie clips, see *MovieClip.onKeyDown( )* .

Note that `focusEnabled` and `tabEnabled` are not synonyms. Even when `focusEnabled` is `false` , a movie clip can be focused with the Tab key when `tabEnabled` is `true` . Conversely, setting `focusEnabled` to `true` does not enable Tab-focusing for the clip; Tab-focusing is controlled independently by `tabEnabled` and `tabIndex` .

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

See Also

MovieClip ._focusrect, *MovieClip* .tabEnabled, *MovieClip* .tabIndex, *TextField* .selectable, *Selection.setFocus( )*

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Chapter 18. ActionScript Language Reference

MovieClip._focusrect Property — Flash 6

disable automatic yellow rectangle around focused movie clips — *read/write*

mc._focusrect

Description

When a movie clip has keyboard focus, Flash displays a yellow rectangle around it, which is not always aesthetically desirable. You can turn off the yellow rectangle for an individual movie clip by setting its _focusrect property to false:

```
listbox_mc._focusrect = false;
```

When _focusrect is true and *mc* has keyboard focus, Flash displays the yellow rectangle and the contents of *mc*'s _over frame (if a frame labeled _over exists). When _focusrect is set to false, Flash displays just the _over frame. When _focusrect is null (the default), the value of the global _focusrect property is used. To disable the yellow focus rectangle for all movie clip and buttons in a movie, set the global _focusrect property to false. Note that the value of the global _focusrect property is not reflected by the _focusrect property of individual movie clip instances. However, setting an individual movie clip's _focusrect property overrides the global setting.

Usage

The Tab key does not work reliably in Test Movie mode unless Control Disable Keyboard Shortcuts is enabled. Be sure to test keyboard behavior in the Standalone Player, if applicable, and in the Flash Player for all targeted web browsers.

See Also

Button._focusrect, _focusrect

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Chapter 18. ActionScript Language Reference

MovieClip._framesloaded Property — Flash 4

the number of frames of a clip or movie that have downloaded to the Player — ***read-only***

mc._framesloaded

Description

The integer _framesloaded property indicates how many frames of *mc* have been loaded into the Player. It is normally used to create preloaders that pause playback until a sufficient number of frames have downloaded. For a movie clip, the _framesloaded property will equal _totalframes (because clips are loaded in their entirety before playing) unless the instance is in the process of loading an external *.swf* file due to a *loadMovie( )* invocation. Therefore, the _framesloaded property is useful only with main movies or external *.swf* files loading into instances or levels. To monitor load progress in bytes rather than frames, use *getBytesLoaded( )* and *getBytesTotal( )*.

Preloader code is traditionally placed directly on the main timeline of the movie being preloaded. A simple approach is to loop between frames 1 and 2 until the movie has loaded, at which point we send the playhead to the movie's start frame. For example:

```
// CODE ON FRAME 1
if (_framesloaded > 0 && _framesloaded =  = _totalframes) {
  gotoAndPlay("beginMovie");
}

// CODE ON FRAME 2
gotoAndPlay(1);
```

As of Flash 6, we can alternatively use the *onEnterFrame( )* movie clip event handler to build a more portable preloader. In the movie we wish to preload, at the frame where we want preloading to start, we invoke the *stop( )* function. Then, on that movie's timeline, we create a movie clip with the following *onEnterFrame( )* handler:

```
stop();
this.createEmptyMovieClip("loader", 1);
loader.onEnterFrame = function () {
  this.loaded = this._parent._framesloaded;
  if (this.loaded > 0 && this.loaded =  = this._parent._totalframes) {
    this._parent.gotoAndPlay("beginMovie");
    this.removeMovieClip();
  }
}
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

In this example, the clip tracks its parent's load progress and starts its parent playing when loading has finished. By using a movie clip as a preloader, we avoid the need for a loop on the preloading movie's timeline. A movie clip preloader can even be turned into a component, providing easier workflow for less experienced developers.

Notice that in our preloader examples we checked whether `_framesloaded > 0` in addition to whether `_framesloaded = = _totalframes`. This is necessary because when a movie is unloaded from a clip, that clip has a `_totalframes` of 0. Hence, if `_framesloaded` is 0 (as it might be on a very slow connection), the comparison `_framesloaded = = _totalframes` can return `true` even when no frames have yet loaded. Our check prevents the movie from skipping ahead before the appropriate content has loaded. This precaution is not necessary with preloaders placed on the main timeline of loading *.swf* files, because their `_totalframes` property is never 0.

The *loadMovie( )* and *unloadMovie( )* functions are executed asynchronously. Hence, the value of `_framesloaded` varies according to when it is called and the value of *`mc`*, as follows:

- If *`mc`* is `undefined`, `_framesloaded` returns `undefined`.
- After a *loadMovie( )* or *unloadMovie( )* statement occurs in source code, but before it executes, and before any of the external file has been retrieved, `_framesloaded` returns the number of frames in the old *`mc`* (that existed prior to the *loadMovie( )* or *unloadMovie( )* statement).
- Once part of the new *.swf* file has been retrieved, `_framesloaded` returns the number of frames loaded so far, from 0 to `_totalframes`, unless the file at *`url`* is not found.
- If the file at *`url`* is not found and *`mc`* is a movie clip, `_framesloaded` returns -1 and *`mc`*`._totalframes` stores 0. However, if *`mc`* is a document `_level`, then `_framesloaded` and `_totalframes` both return `undefined`.
- After an *unloadMovie( )* command executes, `_framesloaded` returns 0 if *`mc`* is a movie clip, and it returns `undefined` if *`mc`* is a document level, such as `_level0`.

Example

Preloaders often include a horizontal loading bar and a text field indicating the percentage of a movie that has downloaded. Loading bars are implemented as clips, resized with either the `_width` property or the `_xscale` property. However, note that a clip scales proportionately on the right and left sides of its registration point. Therefore, to resize a clip from one side only, we must place all of the clip's content on one side of the registration point in the clip's symbol. The following example shows how to add a loading bar and a status text field to our preceding clip handler code. It draws the preload bar using the *MovieClip* Drawing API, but the loading code would also work with manually drawn movie clips.

```
// A Portable Preloader with a Status Bar
// Stop the loading movie
stop();
// Create a container clip to do the preloading
this.createEmptyMovieClip("loader", 1);
// Create load bar clips
this.loader.createEmptyMovieClip("totalBar", 2);
this.loader.createEmptyMovieClip("loadedBar", 3);
// Create a text field for output
this.loader.createTextField("loadProgress_txt", 4, 80, 10, 200, 20);
// In the load bar clips, draw a red line on top of a thicker black line
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

```
this.loader.loadedBar.lineStyle(5, 0xFF0000);
this.loader.totalBar.lineStyle(8, 0x000000);
this.loader.loadedBar.lineTo(250, 0);
this.loader.totalBar.lineTo(250, 0);
// Assign the onEnterFrame() handler that will do the preloading work
this.loader.onEnterFrame = function () {
  // Determine the percentage of bytes that have loaded
  this.percentDone = Math.floor((this._parent.getBytesLoaded()
                          / this._parent.getBytesTotal()) * 100);
  // Display the percentage of loaded bytes in our text field
  this.loadProgress_txt.text =  this.percentDone + "% complete";
  // Set the size of our loadedBar clip
  this.loadedBar._xscale = this.percentDone;
  // Check how many frames have loaded
  this.loaded = this._parent._framesloaded;
  // If loading is complete...
  if (this.loaded > 0 && this.loaded =  = this._parent._totalframes) {
    // ...start the movie, and...
    this._parent.gotoAndPlay("beginMovie");
    // ...delete the loader.
    this.removeMovieClip();
  }
}
```

Use the View Bandwidth Profiler command in Test Movie mode to simulate movie download for preloader testing.

See Also

MovieClip.getBytesLoaded( ), *MovieClip.onData( )*, `MovieClip._totalframes`; "The Bandwidth Profiler," in the online technote http://www.moock.org/asdg/technotes

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

MovieClip.getBounds( ) Method Flash 5

determine the bounding box of a clip or movie

```
mc.getBounds(targetCoordinateSpace)
```

Arguments

targetCoordinateSpace

A string indicating the path to the movie or clip in whose coordinate space `mc`'s dimensions are measured. Because a movie clip reference is converted to a path when used in a string context, `targetCoordinateSpace` can also be a movie clip object reference, as in *`mc`.getBounds(`_root`)* versus *`mc`.getBounds(`"_root"`)*. Defaults to `mc` if not specified.

Returns

An object whose properties—`xMin`, `xMax`, `yMin`, and `yMax`—describe the bounding box of the space occupied by `mc`. These four properties of the object specify the leftmost, rightmost, topmost, and bottommost pixel coordinates of `mc`.

Description

The *getBounds( )* method returns an anonymous object with properties that define the rectangular area occupied by `mc` (i.e., `mc`'s *bounding box*). To retrieve the values stored in the returned object, we must access that object's properties, as shown in the following Example.

The dimensions of the bounding box of a clip can be measured relative to any other clip or movie. Using the `targetCoordinateSpace` argument, we can pose the question, "If `mc` resided on `targetCoordinateSpace`'s canvas, what area would it occupy?" The answer differs depending on whether `targetCoordinateSpace` is a main movie or a clip instance; the origin point of the main movie's coordinate space is the top-left corner of the Stage, but the origin point of an instance's coordinate space is its registration point, as marked in the clip's Library symbol (where it is shown as a crosshair).

The *getBounds( )* method can be used to perform basic collision detection between a movie or clip and some other point (though *MovieClip.hitTest( )* serves this purpose better). It might also be used to identify a rectangular region in which to place a clip added to a movie with *MovieClip.attachMovie( )*.

Example

```
// Retrieve the bounds for the current clip
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

```
var clipBounds = this.getBounds();
// Store the bounds in corresponding variables
var leftX   = clipBounds.xMin;
var rightX  = clipBounds.xMax;
var topY    = clipBounds.yMin;
var bottomY = clipBounds.yMax;
```

See Also

MovieClip.globalToLocal( ), MovieClip.hitTest( ), MovieClip.localToGlobal

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Chapter 18. ActionScript Language Reference

MovieClip.getBytesLoaded( ) Method Flash 5

check the number of bytes that have downloaded to the Player

```
mc.getBytesLoaded()
```

Returns

An integer representing the number of bytes of *mc* that have finished downloading to the Player. Divide by 1024 to conv to kilobytes (KB).

Description

The *getBytesLoaded ( )* method tells us the number of bytes of a movie or JPEG image that have downloaded into the Fla However, data usually arrives in chunks of several hundred or several thousand bytes, and *getBytesLoaded ( )* measures b terms of these chunks only. So, if a movie's first frame is 200 bytes in size and its second frame is 3000 bytes in size, *getBytesLoaded ( )* may return 200 and 3200, but it will never return any increment in between.

Note that internal movie clips are always loaded entirely before they are displayed, so the return value of *getBytesLoaded* internal movie clip is always the same as *getBytesTotal ( )* , unless the movie clip is currently loading an external file in re *loadMovie ( )* . Therefore, *getBytesLoaded ( )* is generally useful only when used with an external *.swf* or *.jpg* file being l into an instance or level.

The *loadMovie ( )* and *unloadMovie ( )* functions are executed asynchronously. Hence, the return value of *getBytesLoade* varies according to when it is called and the value of *mc* , as follows:

- If *mc* is `undefined` , *getBytesLoaded( )* returns `undefined` .
- After a *loadMovie( )* or *unloadMovie( )* statement occurs in source code, but before it executes, and before any of th external file has been retrieved, *getBytesLoaded( )* returns the size of the existing content in *mc* (prior to the *loadM* or *unloadMovie( )* statement). An empty movie clip created during authoring has a byte size of 4. An empty movie created via *MovieClip.createEmptyMovieClip( )* has a byte size of 0.
- Once bytes have been retrieved, *getBytesLoaded( )* returns the number of bytes loaded so far, unless the file at *url* found.
- If the file at *url* is not found, *getBytesLoaded( )* returns 0 (even if *mc* is a document `_level` that did not prev exist). However, *getBytesTotal( )* returns -1.
- After an *unloadMovie( )* command executes, *getBytesLoaded( )* returns 0 if *mc* is a movie clip, or it returns `unde` if *mc* is a document level, such as `_level0` . The same applies to *getBytesTotal( )* .

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Like `_framesloaded` , *getBytesLoaded ( )* is normally used to build preloaders. It can be used in concert with *getByt ( )* to create a more accurate progress bar than is possible with `_framesloaded` and `_totalframes` (because the by each frame may not be equal—a movie with 10 frames is not 30% loaded when three frames are loaded if the frames diffe widely in size). See `_framesloaded` for example preloader code based on the number of downloaded frames.

Usage

Unfortunately, in Flash 6, the return value of *getBytesLoaded ( )* reflects the uncompressed size of Flash *.swf* files being not the size of the file after Flash compression has been applied. This means that *getBytesLoaded ( )* for a *.swf* file with a compression size of 2,000 bytes might return, say, 10,000 bytes.

Example

The following code shows a simple preloader that displays download progress in two text fields:

```
// CODE ON FRAME 1 OF A MOVIE'S MAIN TIMELINE
if (this._framesloaded > 0 && this._framesloaded =  = this._totalframes) {
  this.gotoAndPlay("beginMovie");
} else {
  // Show the load progress in text fields. Divide by 1024 to convert bytes
  loadProgressOutput_txt.text = Math.floor(this.getBytesLoaded()/1024);
  loadTotalOutput_txt.text = Math.floor(this.getBytesTotal()/1024);
}

// CODE ON FRAME 2
this.gotoAndPlay(1);
```

See Also

loadMovie( ) , `MovieClip` `._framesloaded` , *MovieClip.getBytesTotal( )* , *MovieClip.loadMovie( )* , `MovieC` `._totalframes` , *MovieClip.unloadMovie ( )* , *unloadMovie ( )*

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

MovieClip.getBytesTotal( ) Method Flash 5

check the size of a clip or movie, in bytes

```
mc.getBytesTotal()
```

Returns

An integer representing the uncompressed size of `mc`, in bytes. Divide by 1024 to convert bytes to kilobytes (KB).

Description

The *getBytesTotal( )* method tells us the uncompressed size, in bytes, of a clip instance or the main movie. When invoked on the main movie, *getBytesTotal( )* reports the size of the entire *.swf* file. It is normally used in concert with *getBytesLoaded( )* to create preloaders for main movies and *.swf* or *.jpg* files loaded into instances or levels via *loadMovie( )*.

The *loadMovie( )* and *unloadMovie( )* functions are executed asynchronously. Hence, the return value of *getBytesTotal( )* varies according to when it is called and the value of `mc`, as follows:

- If `mc` is `undefined`, *getBytesTotal( )* returns `undefined`.
- After a *loadMovie( )* or *unloadMovie( )* statement occurs in source code, but before it executes, and before the size of the external file has been determined, *getBytesTotal( )* returns the size of the existing content in `mc` (prior to the *loadMovie( )* or *unloadMovie( )* statement). An empty movie clip created manually during authoring has a byte size of 4. An empty movie clip created via *MovieClip.createEmptyMovieClip( )* has a byte size of 0.
- Once the file size is determined, *getBytesTotal( )* returns the complete size of `mc`, unless the file at `url` is not found.
- If the file at `url` is not found, *getBytesTotal( )* returns -1 (even if `mc` is a document `_level` that did not previously exist) and *getBytesLoaded( )* returns 0.
- After an *unloadMovie( )* command executes, *getBytesTotal( )* returns 0 if `mc` is a movie clip or `undefined` if `mc` is a document level, such as `_level0`. The same applies to *getBytesLoaded( )*.

Usage

As with *getBytesLoaded( )*, in Flash 6 the return of *getBytesTotal( )* reflects the uncompressed size of the *.swf*

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

file being loaded, not the size of the file after Flash compression has been applied. This means that *getBytesTotal( )* for a *.swf* file with a post-compression size of 2,000 bytes might return, say, 10,000 bytes. As such, it cannot be used to predict the time left in the download of a compressed *.swf* file. It is intended to offer only a relative measure of the percentage of the file that is downloaded, in concert with *getBytesLoaded( )*.

See Also

loadMovie( ), `MovieClip._framesloaded`, *MovieClip.getBytesLoaded( )*, *MovieClip.loadMovie( )*, `MovieClip._totalframes`, *MovieClip.unloadMovie( )*, *unloadMovie( )*

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Chapter 18. ActionScript Language Reference

MovieClip.getDepth( ) Method Flash 6

returns the movie clip's position in the visual content stack

```
mc.getDepth()
```

Returns

An integer specifying the depth of *mc* in its parent's visual content stack.

Description

The *getDepth ( )* method returns the integer position of *mc* in its parent movie clip's content stack. Higher depths obscure lower depths, so a depth of 6 is in front of 5, which is in front of 4, and so on. Depth positions range from -16384 to 1048575. Depths above -1 are reserved for dynamically generated content; depths from -16383 to -1 (inclusive) are reserved for author-time content; and depth -16384 is reserved for dynamic content that appears beneath all author-time content in each clip (e.g., a runtime background).

Each instance's depth is initialized when the clip is created (either in the Flash authoring tool or via *attachMovie ( )* , *createEmptyMovieClip ( )* , or *duplicateMovieClip ( )*). However, an instance can be moved to a different depth via *swapDepths ( )* . Content placed at a given depth replaces any content previously loaded at that depth.

For important details, see Section 13.4.

Example

The following code creates three clips and draws a line in each. The first line clip (red) is created behind the second line clip (blue). Then the third line clip (green) is placed in the red line clip's depth, effectively replacing it. The `redline_mc` clip's depth is dynamically determined via *getDepth( )* .

```
// Red line appears in back (depth 1)
this.createEmptyMovieClip("redline_mc", 1);
redline_mc.lineStyle(10, 0xFF0000);
redline_mc.lineTo(100, 10);

// Blue line appears in front (depth 2)
this.createEmptyMovieClip("blueline_mc", 2);
blueline_mc.lineStyle(10, 0x0000FF);
blueline_mc.lineTo(100, 0);

// Green line replaces red line, because it was created at redline_mc's dept
this.createEmptyMovieClip("greenline_mc", redline_mc.getDepth());
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

```
greenline_mc.lineStyle(10, 0x00FF00);
greenline_mc.lineTo(100, 10);
```

See Also

MovieClip.swapDepths( ) ; Section 13.4

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

MovieClip.getURL( ) Method — Flash 5

load a document into a browser window

```
mc.getURL(url)
mc.getURL(url, window)
mc.getURL(url, window, method)
```

Arguments

url

A string specifying the location of the document to load or external script to run.

window

An optional string, specifying the name of the browser window or frame into which to load the document. Can be a custom name or one of the four presets: "`_blank`", "`_ parent`", "`_self`", or "`_top`".

method

An optional string literal specifying the method by which to send the variables from `mc` to an external script. Must be either the literal "`GET`" or "`POST`"; no other expression is allowed.

Description

The *MovieClip.getURL( )* method is an alternative to the global *getURL( )* function. This *MovieClip* method form is useful only if variables are being sent via `url` to a server-side script, in which case *getURL( )* sends the variables from `mc`, which does not have to be the current timeline. For general usage instructions, see the global *getURL( )* function. For an object-oriented means of sending variables to a server, see *LoadVars.send( )*.

See Also

getURL( ), LoadVars.send( )

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

MovieClip.globalToLocal( ) Method Flash 5

convert a point on the main Stage to clip coordinates

```
mc.globalToLocal(point)
```

Arguments

point

A reference to an object with two properties, `x` and `y` , that describe a point on the main Stage of the Player (i.e., o `_level0`). Both `x` and `y` can be any floating-point number.

Description

The *globalToLocal ( )* method converts the `x` and `y` properties of `point` from coordinates on the main Stage to coordinates in the coordinate space of `mc` . Note that *globalToLocal ( )* does not return a new object, it merely modifies the existing `x` and `y` values of `point` . It has no effect when invoked on a main movie object (i.e., `_root`), because th original and target coordinate spaces are identical.

To use *globalToLocal ( )* , we must first create a so-called *point* or *coordinates* object with `x` and `y` properties. To do so, create a generic object from the *Object* class, and add `x` and `y` properties to it. For example:

```
var myPoint = new Object();
myPoint.x = 10;
myPoint.y = 20;
```

The `x` and `y` properties of our object are positions on the horizontal and vertical axes of the main Stage, relative to its top-left corner. For example, an `x` of 10 is 10 pixels to the right of the Stage's left edge, and a `y` of 20 is 20 pixels below the Stage's top border. With our object created and our `x` and `y` properties set, we then pass the object to the *globalToLocal (* method, as in:

```
myClip_mc.globalToLocal(myPoint);
```

When *globalToLocal ( )* is executed, the values of `myPoint` 's `x` and `y` properties are transformed to represent a point in the space of `myClip_mc` , measured from its registration point. By examining the new values of our `myPoint` object's properties, we answer the question, "Where does the point *(x, y)* of the main Stage appear in `myClip` ?" For example:

```
xInClip = myPoint.x;
yInClip = myPoint.y;
```

Example

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

The following example calculates the offset from the upper-left corner of the main Stage to the registration point of the current clip:

```
pt = new Object();          // Create generic object to hold our point
pt.x = 0;                    // Left border of main Stage
pt.y = 0;                    // Top border of main Stage
this.globalToLocal(pt);      // Convert pt to local coordinates

trace("From the current clip, the top-left corner of the main Stage is at ")
trace("x: " + pt.x + "y: " + pt.y);
```

See Also

MovieClip.localToGlobal

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Chapter 18. ActionScript Language Reference

MovieClip.gotoAndPlay( ) Method Flash 5

jump to a given frame, then play the movie or clip

```
mc.gotoAndPlay(frameNumber)
mc.gotoAndPlay(frameLabel)
```

Arguments

frameNumber

A positive integer indicating the number of the frame to which the playhead of `mc` should jump before playing. If `frameNumber` is less than 1, the call to *MovieClip.gotoAndPlay( )* is ignored. Note that this behavior differs from the global version of *gotoAndPlay( )*, where a value less than 1 is treated as 1. If `frameNumber` is greater than the number of frames in `mc`'s timeline, the playhead is sent to the last frame.

frameLabel

A string indicating the label of the frame to which the playhead of `mc` should jump before playing. If `frameLabel` is not found, the operation fails silently. This behavior differs from the global version of *gotoAndPlay( )*, where a missing frame label is treated as frame 1.

Description

The *MovieClip.gotoAndPlay( )* method is an alternative to the global *gotoAndPlay( )* function. Use the *MovieClip* method form to control remote movie clips or movies, specified by *mc*.

For general usage instructions, see the global *gotoAndPlay( )* function.

Example

```
// Send the part1_mc clip to the label intro, then play part1_mc
part1_mc.gotoAndPlay("intro");
```

See Also

gotoAndPlay( ), MovieClip.gotoAndStop( )

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Chapter 18. ActionScript Language Reference

MovieClip.gotoAndStop( ) Method Flash 5

jump to a given frame, and stop the playhead

```
mc.gotoAndStop(frameNumber)
mc.gotoAndStop(frameLabel)
```

Arguments

frameNumber

A positive integer indicating the number of the frame to which the playhead of `mc` should jump. If `frameNumber` is less than 1, the call to *MovieClip.gotoAndStop( )* is ignored. Note that this behavior differs from the global version of *gotoAndStop( )*, where a value less than 1 is treated as 1. If `frameNumber` is greater than the number of frames in `mc`'s timeline, the playhead is sent to the last frame.

frameLabel

A string indicating the label of the frame to which the playhead of `mc` should jump. If `frameLabel` is not found, the operation fails silently. This behavior differs from the global version of *gotoAndStop( )*, where a missing frame label is treated as frame 1.

Description

The *MovieClip.gotoAndStop( )* method is an alternative to the global *gotoAndStop( )* function. Use the *MovieClip* method form to control remote movie clips or main movies specified by *mc*.

For general usage instructions, see the global *gotoAndStop( )* function.

Example

```
// Send the mainMenu_mc clip to frame 6, and stop the playhead there
mainMenu_mc.gotoAndStop(6);
```

See Also

gotoAndStop( ), MovieClip.gotoAndPlay( )

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Chapter 18. ActionScript Language Reference

MovieClip._height Property Flash 5

height of a clip or movie, in pixelsFlash 4; enhanced in Flash 5read-only in Flash 4; read/write in

mc._height

Description

The floating-point _height property is a nonnegative number specifying the height of *mc* , in pixels. If *mc* has no content, _height is 0. The _height property measures the content of a clip as the distance between the topmost occupied pixel and the bottommost occupied pixel, even if there is empty space between those pixels. An *occupied pixel* i pixel that contains a shape, graphic, button, movie clip, text field or other content element. Changes made to a clip's heigh the authoring tool or via _yscale are reflected by _height .

Setting the value of _height resizes a movie clip vertically. Attempts to set _height to a negative value are ignored. Setting a clip's _height to 0 does not make it invisible; rather, the clip becomes a one-pixel horizontal line. Flash's basi unit of measure is a twentieth of a pixel (a "twip"), so the smallest legal _height change is .05 pixels. Smaller increme are ignored:

```
trace(ball_mc._height);   // Displays: 100.
ball_mc._height += .01;
trace(ball_mc._height);   // Still displays 100. The assignment had no effec
```

The _height of the main movie (either _root._height or _level*n* ._height) is not the Stage height as specified under Modify Movie Dimensions in the authoring tool, but rather the height of the contents of the mai movie. As of Flash 6, the movie's specified dimensions are available via Stage.height and Stage.width .

For a loading movie, the _height property returns undefined until the movie has completely loaded.

Usage

Note that when we set the height of a movie clip, lines are scaled proportionally, losing their original point size as set in th Stroke panel. However, the point size of lines set to Hairline in the Stroke panel is not scaled when a movie clip is resized Use hairlines to prevent your strokes from appearing fat or distorted when a clip is scaled.

Example

```
ball_mc._height = 20;     // Set the height of ball_mc to 20 pixels
ball_mc._height /= 2;     // Reduce the height of ball_mc by a factor of 2
```

See Also

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

MovieClip ._width,*MovieClip* ._yscale,Stage.height

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

MovieClip.hitArea Property — Flash 6

assigns the hotspot for a button-enabled movie clip — *read/write*

`mc.hitArea`

Description

The `hitArea` property is used only with movie clips that define button event handlers (e.g., *onPress( )*, *onRelease( )*, etc.). It sets *`mc`*'s button hit area—the physical region that must be under the mouse pointer in order for the button to be activated by a mouseclick. When the value of `hitArea` is `null` or `undefined`, the shape of *`mc`*'s content is used as the hit area. When the value of `hitArea` is a reference to a movie clip, the shape of that clip's content is used as *`mc`*'s hit area. The `hitArea` clip can move, be invisible, or even be reassigned at any time, allowing great flexibility for dynamically generated buttons. For example, an animated button can have a hit area that changes size with the animation, or a button can have a different hit area for its Over, Up, and Down states.

The movie clip used as the `hitArea` for *`mc`* can reside on any clip's timeline, including *`mc`*'s. For example:

```
// Set button_mc's hit area to a clip inside itself
button_mc.hitArea = button_mc.shape_mc;
// Set button_mc's hit area to a clip outside itself
button_mc.hitArea = _root.shapeClips.shape_mc;
```

When the mouse pointer moves into the hit area, it changes from an arrow pointer to a hand pointer. To suppress the display of the hand pointer, set `useHandCursor` to `false`.

For information on implementing button behavior for a movie clip, see the *MovieClip* class, *MovieClip.hitTest( )*, and Section 13.9.

Example

The following code creates a button-enabled movie clip and draws its hit area entirely from scratch. It automatically adjusts the size of the hit area to be twice the size of the text in the button. By defining the `hitArea` property explicitly, we prevent any future content added to the button from being included in the hit area.

```
// Make a clip that will act as a button
this.createEmptyMovieClip("button_mc", 1);

// Add some text to the clip
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

```
button_mc.createTextField("button_txt", 1, 0, 0, 0, 0);
button_mc.button_txt.autoSize = true;
button_mc.button_txt.text = "click here!";
button_mc.button_txt.selectable = false;

// Define a button handler for the button clip
button_mc.onPress = function () {
  trace("thank you for clicking");
}

// Make a clip to be the button clip's hit area, and
// put it inside the button clip
button_mc.createEmptyMovieClip("hitarea_mc", 2);
// Set the hit area rectangle to be twice the size of the button text
var x1 = -button_mc.button_txt._width / 2;
var y1 = -button_mc.button_txt._height / 2;
var x2 = x1 + button_mc.button_txt._width * 2;
var y2 = y1 + button_mc.button_txt._height * 2;
// Draw the hit area rectangle
button_mc.hitarea_mc.moveTo(x1, y1);
button_mc.hitarea_mc.beginFill(0x000000);
button_mc.hitarea_mc.lineTo(x2, y1);
button_mc.hitarea_mc.lineTo(x2, y2);
button_mc.hitarea_mc.lineTo(x1, y2);
button_mc.hitarea_mc.lineTo(x1, y1);
button_mc.hitarea_mc.endFill();

// Make the hit clip invisible
button_mc.hitarea_mc._visible = false;

// Assign the drawn clip as the button's hit area
button_mc.hitArea = button_mc.hitarea_mc;
```

See Also

The *Button* class, *MovieClip*.enabled, *MovieClip.hitTest( )*, *MovieClip*.useHandCursor; Section 13.9

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Chapter 18. ActionScript Language Reference

MovieClip.hitTest( ) Method Flash 5

check if a point or another clip intercept a given clip

```
mc.hitTest(x, y, shapeFlag)
mc.hitTest(target)
```

Arguments

x

The horizontal coordinate of the point to test.

y

The vertical coordinate of the point to test.

shapeFlag

An optional Boolean value indicating whether the function should test for intersection of the point (`x`, `y`) against the bounding box of `mc` (`false`) or against the actual pixels of `mc` (`true`). Defaults to `false` if not supplied. Note that `shapeFlag` can be used only with the `x` and `y` argument syntax, not with the `target` argument. It is meaningful only when `mc` has an irregular contour or a hole like a doughnut; it has no impact if `mc` is a solid rectangular object.

target

A string indicating the path to the movie clip to test for collision with `mc`. Because movie clip references are converted to paths when used in a string context, `target` can also be a movie clip object reference, as in `mc`.*hitTest(*`ball`*)* versus `mc`.*hitTest(*`"ball"`*)*.

Returns

A Boolean indicating the result of the collision-detection test. The result is `true` under any of the following circumstances:

- The point *(x,y)* on the main Stage intersects with any occupied pixel of `mc`. An occupied pixel is one that contains a visual element, such as a shape or text.
- The `shapeFlag` property is `false` and the point *(x,y)* on the main Stage intersects with any pixel in the bounding box of `mc`. The bounding box of `mc` is the smallest rectangle that can encompass every occupied pixel of `mc` (as returned by *MovieClip.getBounds( )*).
- Any pixel in the bounding box of `target` intersects any pixel in the bounding box of `mc`.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

The result is `false` under any of the following circumstances:

- The *`shapeFlag`* property is `true` and the point *(x,y)* on the main Stage does not intersect with any occupied pixel of *`mc`*. An occupied pixel is one that contains a visual element, such as a shape or text.
- The point *(x,y)* on the main Stage does not intersect with any pixel in the bounding box of *`mc`*. The bounding box of *`mc`* is the smallest rectangle that can encompass every occupied pixel of *`mc`*.
- No pixels in the bounding box of *`target`* intersect with any pixel in the bounding box of *`mc`*.

Description

The *hitTest( )* method is used to determine whether a movie clip or specific point intersects with (i.e., "hits") *`mc`*.

When checking to see if a point intersects *`mc`*, we provide *hitTest( )* with the *`x`* and *`y`* coordinates of the point to check (relative to the main Stage). We can also provide the optional *`shapeFlag`* argument, which indicates whether the collision test should use the actual pixels of *`mc`* or just *`mc`*'s bounding box (the smallest rectangle that can encompass every occupied pixel of *`mc`*). Checking the actual pixels of *`mc`* allows us to determine whether the point *(x, y)* is an occupied pixel inside the contour of *`mc`*, not merely whether it is any point inside *`mc`*'s bounding box.

When we're checking to see if a movie clip intersects *`mc`*, we provide *hitTest( )* with the *`target`* argument specifying the clip to check against. Collision detection between *`target`* and *`mc`* always uses the bounding boxes of the clips; *hitTest( )* does not support pixel-perfect clip-versus-clip detection. Manual pixel-perfect collision-detection routines can be difficult to create and processor-intensive to run. In many situations—for example, a simple spaceship game—it's common practice to detect against a bounding circle rather than exact pixels (see the following Example).

The return value of *hitTest( )* is unaffected by the `_visible` or `_alpha` properties, which can hide a movie clip from view. An invisible movie clip can still be hit-tested against a point or another invisible clip.

Usage

Note that collision is always tested relative to the location of *`mc`* on the main Stage of the Player. Therefore, when *hitTest( )* is used with a single point, the arguments `x` and `y` should always describe a point using Stage coordinates. See *MovieClip.localToGlobal* for information on converting clip coordinates to Stage coordinates. In clip-versus-clip detection, the coordinates of clips on different timelines are automatically converted to global (i.e., Stage) coordinates.

Example

This example shows how to detect the intersection of two circles manually, without using *hitTest( )*:

```
// Check the distance between two circular clips on both axes.
var deltaX = clip1._x - clip2._x;  // Horizontal distance
var deltaY = clip1._y - clip2._y;  // Vertical distance

// Store the radius of each circle in a convenient property.
var radius1 = clip1._width / 2;
var radius2 = clip2._width / 2;
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

```
// If the distance between the circles' centers squared is less
// than or equal to the total length of the two radii squared,
// an intersection occurs.
if ((deltaX * deltaX) + (deltaY * deltaY)
    <= (radius1 + radius2) * (radius1 + radius2)) {
  trace("intersecting");
} else {
  trace("not intersecting");
}
```

Here we check whether `paddle_mc` intersects `ball_mc` using *hitTest( )*:

```
if (paddle_mc.hitTest("ball_mc")) {
  trace("The paddle hit the ball.");
}
```

Here we check whether the mouse pointer is over an occupied pixel within `house_mc`'s contour. Notice that the coordinates of the pointer are given in relation to `_root` (the main Stage). Gaps in a movie clip are detectable when *`shapeFlag`* is `true`; for example, if the `house_mc` clip uses empty space to represent its windows, then *hitTest( )* will return `false` when the pointer is over a window:

```
if (house_mc.hitTest(_root._xmouse, _root._ymouse, true)) {
  trace("You're pointing to the house.");
}
```

See Also

MovieClip.getBounds( ), *MovieClip.localToGlobal( )*

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

MovieClip.lineStyle( ) Method — Flash 6

sets stroke thickness, color, and transparency for drawing at runtime

```
mc.lineStyle(thickness, RGB, alpha)
```

Arguments

thickness

The integer line point size, ranging from 0 to 255, where 0 is hairline. If not specified, lines are not drawn. Values below 0 are interpreted as 0; those above 255 are interpreted as 255.

RGB

The integer color of the line, normally supplied in RGB hexadecimal triplet notation—`0x`*RRGGBB*, where *RR*, *GG*, and *BB* are two-digit hex numbers representing Red, Green, and Blue. For example, the hex integer `0xFF0000` indicates red. Defaults to black if not supplied. For a complete discussion of RGB triplet notation, see the *Color* class.

alpha

An integer between 0 and 100 specifying the opacity (or, conversely, the transparency) of the shape as a percentage—0 is completely transparent, whereas 100 is completely opaque. Defaults to 100 if not supplied. Values above 100 are interpreted as 100; those below 0 are interpreted as 0.

Description

The *lineStyle( )* method sets the stroke thickness, color, and transparency for all lines and curves subsequently drawn in *mc* via the *lineTo( )* and *curveTo( )* methods. By default, no stroke is applied to drawings in *mc*. The *lineStyle( )* method must be invoked explicitly for each new movie clip and after every call to *clear( )*, or no stroke will appear (although filled regions can still be drawn without a stroke).

The following examples are valid ways to modify the line style for `drawing_mc`:

```
drawing_mc.lineStyle()                  // Clears the style
                                        // (lines and curves not stroked)
drawing_mc.lineStyle(1)                 // 1 pt solid black
drawing_mc.lineStyle(1,0x00FF00)        // 1 pt solid green
drawing_mc.lineStyle(1,0x00FF00, 50) // 1 pt semi-transparent green
```

We may change the stroke at any time by calling *lineStyle( )* again. For example, the following code draws a square with progressively thicker lines colored black, red, green, and blue:

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

```
this.createEmptyMovieClip("drawing_mc", 2);

drawing_mc.lineStyle(1, 0x000000, 100);
drawing_mc.lineTo(100, 0);
drawing_mc.lineStyle(5, 0xFF0000, 100);
drawing_mc.lineTo(100, 100);
drawing_mc.lineStyle(10, 0x00FF00, 100);
drawing_mc.lineTo(0, 100);
drawing_mc.lineStyle(15, 0x0000FF, 100);
drawing_mc.lineTo(0, 0);
```

A *thickness* of 0 sets the stroke to hairline (a one-pixel line that does not increase in thickness when the clip or movie is scaled up). To turn the stroke off completely, set thickness to `undefined` or call *lineStyle( )* with no parameters. For example:

```
drawing_mc.lineStyle(undefined);   // Turn off lines in drawing_mc
drawing_mc.lineStyle();            // Same thing
```

Even if a line style is defined for a movie clip, we can move the drawing pen to a new location without drawing a line or curve using *MovieClip.moveTo( )*. When the *clear( )* method is invoked to erase all drawings in a clip, the clip's line style reverts to `undefined` (no line).

See Also

MovieClip.beginFill( ), *MovieClip.beginGradientFill( )*, *MovieClip.clear( )*, *MovieClip.curveTo( )*, *MovieClip.endFill( )*, *MovieClip.lineTo( )*, *MovieClip.moveTo( )*

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

MovieClip.lineTo( ) Method Flash 6

draws a straight line at runtime

```
mc.lineTo(x, y)
```

Arguments

x

The horizontal position of the endpoint of the line to be drawn, as a floating-point number. Measured relative to *mc* registration point.

y

The vertical position of the endpoint of the line to be drawn, as a floating-point number. Measured relative to *mc* registration point.

Description

The *lineTo ( )* method draws a straight line from the current drawing pen position to the point (*x* ,*y*) (for a description of t drawing pen see *MovieClip.moveTo( )*). The most recent call to *lineStyle ( )* determines the stroke characteristics of the l (thickness, color, and alpha). If the *lineStyle ( )* is `undefined` , *lineTo( )* moves the drawing pen position but does not d line on screen (exactly as if *moveTo( )* had been called). The line drawn appears *above* all Drawing API content already in , but *beneath* all other content in *mc* (e.g., movie clips, shapes, text, and buttons placed at authoring time, or content attac runtime via *attachMovie( )* , *duplicateMovieClip( )* , or *createTextField( )*). If either of *lineTo( )* 's arguments is missing, t operation fails silently.

The following code creates a movie clip named `drawing_mc` and draws a 2-point black line downward to the point (0, from the drawing pen's current position (0,0). Until the drawing pen's position is modified by *moveTo ( )* , *lineTo ( )* , or *curveTo ( )* , the drawing pen's position is the registration point of *mc* , or the upper-left corner of the Stage if *mc* is *_r*

```
this.createEmptyMovieClip("drawing_mc", 1);  // Create a clip to draw in
drawing_mc.lineStyle(2);                     // Set the stroke to 2 point, b
drawing_mc.lineTo(0, 100);                   // Draw the line
```

After a line or curve is drawn, the drawing pen remains at the end point of the line or curve. Hence, a continuous jagged li easily be drawn using a series of *lineTo ( )* and/or *curveTo ( )* calls:

```
// Create a clip to draw in
this.createEmptyMovieClip("drawing_mc", 1);

// Set the stroke to 1 point, red
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

```
drawing_mc.lineStyle(1, 0xFF0000);

// Draw a jagged line as a series of line segments
drawing_mc.lineTo(50, 100);
drawing_mc.lineTo(-20, 150);
drawing_mc.lineTo(75, 200);
drawing_mc.lineTo(50, 300);
```

Multiple calls to *lineTo ( )* and/or *curveTo ( )* can be used similarly to draw closed shapes. For details, see *MovieClip.beg ( )* .

Of course, movie clips that contain Drawing API content can be manipulated as usual. For example, this code makes our `drawing_mc` clip spin:

```
drawing_mc.onEnterFrame = function () {
  this._rotation++;
}
```

Reader Exercise: Notice that *lineTo( )* accepts absolute coordinates, not relative coordinates. Write a custom method, *lineToDelta( )* , that accepts relative coordinates. For example, *mc.lineToDelta(5, 10)* would draw a line to a point 5 pixel the right of, and 10 pixels below, the current pen position. (Hint: See Example 18-3 under *MovieClip.moveTo( )* , which the pen's current position.)

See Also

MovieClip.beginFill( ) , MovieClip.beginGradientFill( ) , MovieClip.clear( ) , MovieClip.curveTo( ) , MovieClip.endFill(MovieClip.lineStyle( ) , MovieClip.moveTo( )

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Chapter 18. ActionScript Language Reference

MovieClip.loadMovie( ) Method — Flash 5; Flash 6 adds .jpg loading and security restrictions

load an external .swf file into the Player

```
mc.loadMovie(url)
mc.loadMovie(url, method)
```

Arguments

url

A string specifying the location of the external *.swf* or .jpg file to load.

method

An optional string literal indicating the method by which to send variables to an external script. Must be either the literal "`GET`" or "`POST`"; no other expression is allowed.

Description

The *MovieClip.loadMovie( )* method is an alternative to the global *loadMovie( )* function. When invoked as a *MovieClip* method, *loadMovie( )* does not take a `target` parameter; it loads the *.swf* or *.jpg* from `url` into `mc`. The *MovieClip* method syntax is less prone to user error than its global function counterpart because no `target` parameter is required.

For usage instructions, see the global *loadMovie( )* function.

See Also

loadMovie( ), loadMovieNum( ), MovieClip.getURL( ); Section 13.7

About Safari | Terms of Service | Privacy Policy | Contact Us | Help | Submit a Problem

Copyright © 2002 O'Reilly & Associates, Inc. All rights reserved.
1005 Gravenstein Highway North
Sebastopol, CA 95472

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Chapter 18. ActionScript Language Reference

MovieClip.loadVariables( ) Method Flash 5

retrieve an external set of variables

```
mc.loadVariables(url)
mc.loadVariables(url, method)
```

Arguments

url

A string specifying the path to one of two variable sources: a script that produces variables as output, or a text file containing variables.

method

An optional string literal indicating the method by which to send the variables from `mc` to an external script. If specified, variables are both sent and loaded. If omitted, variables are loaded only. Must be either the literal "`GET`" or "`POST`"; no other expression is allowed.

Description

The *MovieClip.loadVariables( )* method is an alternative to the global *loadVariables( )* function. When invoked as a *MovieClip* method, *loadVariables( )* takes no `target` parameter; it loads the variables at `url` into `mc`. The *MovieClip* method syntax is less prone to user error than its global function counterpart because no `target` parameter is required.

For usage instructions, see the global *loadVariables( )* function. As of Flash 6, the *LoadVars* class is preferred over *MovieClip.loadVariables( )*, as it provides a more flexible and powerful means of importing and exporting variables.

See Also

loadVariables( ), the *LoadVars* class; Section 13.7

About Safari | Terms of Service | Privacy Policy | Contact Us | Help | Submit a Problem

Copyright © 2002 O'Reilly & Associates, Inc. All rights reserved.
1005 Gravenstein Highway North
Sebastopol, CA 95472

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

MovieClip.localToGlobal( ) Method Flash 5

convert a point in a clip to main Stage coordinates

```
mc.localToGlobal(point)
```

Arguments

point

A reference to an object with two properties, `x` and `y` , that describe a point in *`mc`* 's coordinate space. Both `x` and can be any floating-point number.

Description

The *localToGlobal ( )* method converts the `x` and `y` properties of *`point`* from coordinates given in *`mc`* 's coordinate sp to coordinates on the main Stage of the Player. Note that *localToGlobal ( )* does not return a new object; it merely modifie the existing `x` and `y` values of *`point`* . It has no effect when invoked on a main movie object (i.e., `_root`), because th original and target coordinate spaces are identical.

To use *localToGlobal ( )* , we must first create a so-called *point* or *coordinates* object with `x` and `y` properties. To do so, create a generic object from the *Object* class, and add `x` and `y` properties to it:

```
myPoint = new Object();
myPoint.x = 10;
myPoint.y = 20;
```

The `x` and `y` properties of our object are positions on the horizontal and vertical axes of *`mc`* , relative to *`mc`* 's registratio point (shown as a crosshair in *`mc`* 's Library symbol). For example, an `x` of 10 is 10 pixels to the right of *`mc`* 's registrat point, and a `y` of 20 is 20 pixels below *`mc`* 's registration point. With our object created and our `x` and `y` properties set, w then pass the object to the *localToGlobal ( )* method, as in:

```
myClip.localToGlobal(myPoint);
```

When *localToGlobal ( )* is executed, the values of `myPoint` 's `x` and `y` properties are transformed to represent the corresponding point on the main Stage, measured from the Stage's upper-left corner. By examining the new values of our `myPoint` object's properties, we answer the question, "Where does the point *(x,y)* of the movie clip appear on the main Stage?" For example:

```
mainX = myPoint.x;
mainY = myPoint.y;
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Example

The following example determines the location of a clip's registration point, relative to the main Stage:

```
pt = new Object();          // Create generic object to hold our point
pt.x = 0;                    // Horizontal registration point of clip
pt.y = 0;                    // Vertical registration point of clip
this.localToGlobal(pt);      // Convert pt to main Stage coordinates

trace("On the main Stage, the registration point of the current clip is at:
trace("x: " + pt.x + "y: " + pt.y);
```

The *localToGlobal ( )* method can be used to convert a point to Stage coordinates for use with the *hitTest ( )* method, whi expects points to be supplied in the Stage's coordinate space. It can also be used to compare points in two different movie clips using a common coordinate space.

See Also

MovieClip.globalToLocal( ) , MovieClip.hitTest( )

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Chapter 18. ActionScript Language Reference

MovieClip.moveTo( ) Method — Flash 6

moves the drawing pen to a new position without drawing a line

```
mc.moveTo(x, y)
```

Arguments

x

The new horizontal position of the pen, as a floating-point number. Measured relative to `mc` 's registration point.

y

The new vertical position of the pen, as a floating-point number. Measured relative to `mc` 's registration point.

Description

Conceptually, lines and curves are drawn in Flash by a theoretical "drawing pen." For all new movie clips, the pen starts o position (0,0)—the clip's registration point—which is typically at its center and is represented by a crosshair in the movie symbol. As lines and curves are drawn (via *lineTo ( )* and *curveTo ( )*), the pen moves around the movie clip's coordinate at the end point of the last line or curve drawn. The pen's current position always indicates the starting point of the next lir be drawn.

The *moveTo ( )* method lets us arbitrarily set the starting point of a line or curve by "picking up" the drawing pen and mo new position without drawing a line to the specified point.

For example, suppose we want to draw a single straight line from (100,100) to (200,200) in a new movie clip. We must fi pen from (0,0) to (100,100) and then draw a line from there to (200,200), as follows:

```
this.createEmptyMovieClip("drawing_mc", 1); // Create a clip to draw in
drawing_mc.lineStyle(1);                     // Set the stroke to 1 point, bl
drawing_mc.moveTo(100, 100);                // Move the pen without drawing a
drawing_mc.lineTo(200, 200);                // Draw the line (this also moves
```

When the *clear ( )* method is invoked to erase all drawings in a clip, the pen position reverts to (0,0). If either of *moveTo* arguments is missing, the operation fails silently.

Usage

Though Flash does not provide built-in access to the current pen position, the *moveTo ( )* , *lineTo ( )* , *curveTo ( )* , and *cl* methods can be modified to store the pen's horizontal and vertical position in properties, as shown in Example 18-3. Befor

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

each built-in method, we store the old version in a property, such as `origLineTo` . This is considered a best practice, in want to restore the default functionality, although we don't do so here.

Example 18-3. Custom pen position properties

```
MovieClip.prototype.origLineTo = MovieClip.prototype.lineTo;
MovieClip.prototype.lineTo = function (x, y) {
  this.penx = x;
  this.peny = y;
  this.origLineTo(x, y);
};

MovieClip.prototype.origMoveTo = MovieClip.prototype.moveTo;
MovieClip.prototype.moveTo = function (x, y) {
  this.penx = x;
  this.peny = y;
  this.origMoveTo(x, y);
};

MovieClip.prototype.origCurveTo = MovieClip.prototype.curveTo;
MovieClip.prototype.curveTo = function (controlX, controlY, anchorX, anchorY
  this.penx = anchorX;
  this.peny = anchorY;
  this.origCurveTo(controlX, controlY, anchorX, anchorY);
};

MovieClip.prototype.origClear = MovieClip.prototype.clear;
MovieClip.prototype.clear = function () {
  this.penx = 0;
  this.peny = 0;
  this.origClear();
};

// Usage:
this.createEmptyMovieClip("drawing_mc", 1);
drawing_mc.lineStyle(1, 0xFF0000);
drawing_mc.lineTo(50, 100);
trace(drawing_mc.penx);                    // Displays: 50
drawing_mc.moveTo(-20, 150);
trace(drawing_mc.penx);                    // Displays: -20
drawing_mc.curveTo(200, 200, 70, 300);
trace(drawing_mc.peny);                    // Displays: 300
```

See Also

MovieClip.beginFill( ) , MovieClip.beginGradientFill( ) , MovieClip.clear( ) , MovieClip.curveTo( ) , MovieClip.endFill(MovieClip.lineStyle( ) , MovieClip.lineTo( )

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Chapter 18. ActionScript Language Reference

MovieClip._name Property Flash 4

the identifier of a clip instance, as a string ***read/write***

mc._name

Description

The _name property specifies the instance name of a movie clip instance as a string. Because it initially reflects the instance name set in the Property inspector during authoring, the _name property doesn't apply to main movies (the main movie is most easily referred to by the global property _root).

Example

We can use _name to determine whether or not to perform some manipulation of a given clip, as shown in the Example for *MovieClip.onLoad( )*. The _name property can also be used to reassign the identifier of a clip. For example:

```
// Change ball to circle
ball._name = "circle";

// Now control the former ball as circle
circle._x = 500;
```

See Also

MovieClip._target, *targetPath( )*; "Movie Clip Instance Names" in Chapter 13

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Chapter 18. ActionScript Language Reference

MovieClip.nextFrame( ) Method Flash 5

advance a clip or movie's playhead one frame and stop it

```
mc.nextFrame()
```

Description

The *MovieClip.nextFrame( )* method is an alternative to the global *nextFrame( )* function. Use the *MovieClip* method form to control remote movie clips or main movies specified by `mc`.

For general usage instructions, see the global *nextFrame( )* function.

Example

```
// Advance slideshow_mc one frame and stop the playhead
slideshow_mc.nextFrame();
```

See Also

gotoAndStop( ), *MovieClip.gotoAndStop( )*, *MovieClip.prevFrame( )*, *nextFrame( )*

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

MovieClip.onData ( ) Event Handler

Flash 5; callback form introduced in Flash 6

---

handler executed when the clip receives variables, or a portion of a loaded movie

```
mc.onData()

onClipEvent (data) {
  statements
}
```

Description

The *onData ( )* event handler is the callback form (and more modern analogue) of the legacy *onClipEvent (data)* event han The *onData ( )* handler executes when external data is loaded into a movie clip. It can be triggered by two quite different circumstances, according to the kind of data being loaded—either external variables, or a *.swf* or *.jpg* file. Note that as a v loader, the *onData( )* event is made obsolete in Flash 6 by the *LoadVars* class's download monitoring methods. As a movi the *onData ( )* event is (in my experience) too buggy to use in Flash 5, though it appears to be working properly in Flash 6.0.29.0.

When we request a series of variables from a server using *loadVariables( )* , we must wait for them to load completely bef their information. When a movie clip receives the end of a batch of loaded variables, the *onData( )* event is triggered, telli safe to execute code that relies on the variables.

For example, suppose we have a guest book movie in which visitors enter comments that we store on a server. When a use attempts to view a comment, we request it from the server using *loadVariables ( )* . But before we can display the commer must pause at a loading screen until we know that the requested data is available. An *onData ( )* event handler tells us whe variables have loaded, at which point we can safely display the comment to the user.

The following code shows a simplified excerpt of a guest book that loads variables when the user clicks a button. In the ex the button loads two URL-encoded variables from a text file into a movie clip using *loadVariables( )* . The movie clip defi *onData ( )* event handler that executes when the variables have loaded. From inside that handler, we display the values of variables. We know the variables are safe to display, because the code in the handler isn't executed until it's triggered by tl *onData ( )* event (i.e., after the data is received):

```
// CONTENT OF OUR guestbook.txt FILE
name=judith&message=hello

// BUTTON INSIDE OUR CLIP
guestBook_mc.go_btn.onRelease = function {
  this._parent.loadVariables("guestbook.txt");
};
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

```
// HANDLER ON OUR CLIP
guestBook_mc.onData = function {
  trace(name);
  trace(message);
};
```

The second use of the *onData ( )* event relates to the loading of external *.swf* and *.jpg* files into movie clips with the *loadM* function. When a *.swf* file is loaded into a host clip, the file begins playing immediately by default, even if it is only partia loaded. This is not always desirable—sometimes we want to guarantee that all or a certain percentage of a *.swf* has loadec playback begins. We can make that guarantee with an *onData( )* event handler and some preloading code.

The *onData ( )* event occurs each time a host movie clip receives a portion of an external *.swf* file. The definition of what constitutes a "portion" is more complex than you might expect. In order for an *onData( )* event to be triggered, at least one complete new frame of the external *.swf* file must have loaded since either the last *onData ( )* event fired or the *.swf* file st loading. (More than one frame of the *.swf* file may actually have loaded in that amount of time, but one frame is the minir number required to prompt an *onData ( )* event.)

The execution of *onData ( )* event handlers is tied to the rendering of frames in the Player. With every frame rendered, the interpreter checks to see if part of an external *.swf* file has been loaded into a clip that has an *onData ( )* event handler. If external *.swf* file has been loaded into such a clip, and the loaded portion contains at least one new frame, then the *onDatc* handler is executed. This process happens once—and only once—per tick of the frame rate (even while the playhead is stc

Note that because the *onData( )* event happens on a per-frame basis, movies with higher frame rates tend to have smoothe preloaders because they receive more frequent updates on the status of loading *.swf* (or *.jpg*) files.

The exact number of *onData( )* events triggered during a *loadMovie ( )* operation depends on the distribution of content in file being loaded and the speed of the connection. A single-frame *.swf* file, no matter how large, will trigger only one *onD* event. On the other hand, a *.swf* file with 100 frames may trigger up to 100 separate *onData( )* events, depending on the r frame rate, the byte size of each frame, and the speed of the network connection. If the frames are large and the connectio more *onData ( )* events will be triggered (up to a maximum of one per frame). If the frames are small and the connection i fewer *onData ( )* events will be triggered (the entire 100 frames may be transferred between the rendering of two frames i Player, prompting only one *onData ( )* event).

So how do we use an *onData( )* event handler to build a preloader? First and foremost, we must be sure to use the Flash 5- *onClipEvent( )* format of the handler:

```
onClipEvent (data) {
  // preload code
}
```

Using *onClipEvent(data)* instead of *onData ( )* ensures that the handler is not deleted when we load our *.swf* file, because former is attached to a clip at authoring time rather than created as a variable that will be lost when the clip loads its new c Then, whenever an *onData ( )* event occurs due to a *loadMovie ( )* function call, we know that an external *.swf* file downl progress. Therefore, from inside an *onClipEvent(data)* event handler, we can check whether enough of the file has downlc before allowing it to play. We do so using the *getBytesLoaded( )* and *getBytesTotal( )* methods, as shown in the following example. (The `_framesloaded` and `_totalframes` movie clip properties can also be used.)

```
onClipEvent (data) {
    trace("data received");           // The show's about to start!

    // Turn on data-transfer light (indicator)
    _root.transferIndicator.gotoAndStop("on");
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

```
    // If we're done loading, turn off data-transfer light, and let the movi
    if (getBytesTotal() > 0 && getBytesLoaded() =  = getBytesTotal()) {
      _root.transferIndicator.gotoAndStop("off");
      play();
    }

    // Display some loading details in text field variables on the _root
    _root.bytesLoaded = getBytesLoaded();
    _root.bytesTotal = getBytesTotal();
    _
root.clipURL = _url.substring(_url.lastIndexOf("/") + 1, _url.length);
}
```

This example also provides feedback while the movie is loading. Note that the *.swf* file being loaded should have a *stop (* function call on its first frame to prevent it from automatically playing before it is completely downloaded. A variation of example is available at the online Code Depot.

Bugs

In Flash 5 on a high-speed connection, the *onData ( )* event does not always execute when a *.swf* file completely loads. Tl unofficial bug has not been verified by Macromedia, but it appears to be fixed in the release 29 of Flash Player 6 (6.0.29.0

See Also

loadMovie( ) , *loadVariables( )* , the *LoadVars* class, *MovieClip.getBytesLoaded( )* , *MovieClip.getBytesTotal( )* , *Movie ._framesloaded* , *MovieClip ._framestotal* , *MovieClip.onLoad ( )* ; Chapter 10

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Chapter 18. ActionScript Language Reference

MovieClip.onDragOut( ) Event Handler Flash 6

Button event; occurs when mouse is pressed over the clip and then moves off the clip

```
mc.onDragOut()

on (dragOut) {
  statements
}
```

Description

The *onDragOut( )* event handler is identical to the *Button.onDragOut( )* event, except that it can also be applied to movie clip instances (though not main movies such as `_root`, `_level0`, etc.). For details, see *Button.onDragOut( )* and Section 13.9.

See Also

Button.onDragOut( ), MovieClip.onDragOver( ); Chapter 13

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Chapter 18. ActionScript Language Reference

MovieClip.onDragOver( ) Event Handler Flash 6

Button event; occurs when the mouse is pressed over the clip then moves off and back on the clip

```
mc.onDragOver()

on (dragOver) {

on (dragOver) {
  statements

on (dragOver) {
  statements
}
```

Description

The *onDragOver( )* event handler is identical to the *Button.onDragOver( )* event, except that it can also be applied to movie clip instances (though not main movies such as `_root`, `_level0`, etc.). For details, see *Button.onDragOver( )* and Section 13.9.

See Also

Button.onDragOver( ), *MovieClip.onDragOut( )*; Chapter 13

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

MovieClip.onEnterFrame ( ) Event Handler Flash 6

handler executed at every tick of the frame rate in the Flash PlayerFlash 5; callback form introduced in

```
mc.onEnterFrame()

onClipEvent (enterFrame) {
  statements
}
```

Description

The *onEnterFrame( )* event handler is the callback form (and more modern analogue) of the legacy *onClipEvent (enterFrame)* event handler. The *onEnterFrame( )* handler is executed once for every tick of the frame rate of a movie, such as when the playhead advances or when the Player rerenders the current frame while the playhead is stopped. In either case, the frequency of the *onEnterFrame( )* event is directly related to the frame rate achieved by the Player. If the frame rate specified at authoring time under Modify Document is 20 frames per second, then a tick of the frame rate will theoretically occur every 1/20th of a second. But if the Player cannot display the movie at the frame rate specified (for example, because the computer is too slow or has many applications open), the actual frame rate will be lower and a tick will be longer. For example, if the actual frame rate is 12 frames per second, a tick will be 1/12th of a second. Note that a *tick* is not a constant time measurement, nor is it even a formal term in ActionScript (as it is in Lingo); it simply denotes a single cycle of the frame rate in the Flash Player.

The following handler increases `cloud_mc`'s size by 10 pixels per frame-rate tick:

```
cloud_mc.onEnterFrame = function () {
  this._height += 10;
  this._width += 10;
};
```

When displayed in the Flash Player, all Flash movies are constantly running, even when nothing is moving on screen or when a movie's playhead is stopped on a frame. Hence, an individual movie clip's *onEnterFrame( )* handler will be executed repeatedly for as long as that clip is on stage, regardless of whether the clip is playing or stopped. If a clip's playhead is moved by a *gotoAndStop( )* function call, the clip's *onEnterFrame( )* event handler is still triggered with each passing frame. And if every playhead of an entire movie has been halted with a *stop( )* function, all *onEnterFrame( )* event handlers on all clips will still execute.

The *onEnterFrame( )* event normally is used to update the state of a movie clip repeatedly over time. But an *onEnterFrame( )* event handler need not apply directly to the clip that bears it—*onEnterFrame( )* can be used on a movie's main timeline or on a single-frame, empty clip to execute code repeatedly. This technique, called a *clip event loop* (or, more loosely, a *process*) is demonstrated in Section 8.7. The *setInterval( )* function is also a

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

useful tool for repeatedly executing code.

To stop a callback from handling the *onEnterFrame( )* event, use the *delete* operator to remove the handler, as in:

```
// Make sure to omit the () operator after the function name!
delete theClip_mc.onEnterFrame;
```

Note that the code in an *onEnterFrame( )* event handler is executed *before* any code that appears on the timeline of the clip containing the handler.

Example

The following code extends our earlier example to make the `cloud_mc` clip grow to a specific size:

```
cloud_mc.maxHeight = 300

cloud_mc.onEnterFrame = function () {
  if (this._height < this.maxHeight) {
    this._height += 10;
  } else {
    delete this.onEnterFrame;
  }
}
```

See Also

setInterval( ); Section 8.7

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Chapter 18. ActionScript Language Reference

MovieClip.onKeyDown ( ) Event Handler Flash 6

handler executed when a key is depressed while the clip has focusFlash 5; callback form introduced in

```
mc.onKeyDown()

onClipEvent (keyDown) {
  statements
}
```

Description

The *onKeyDown( )* event handler is the callback form (and more modern analogue) of the legacy *onClipEvent(keyDown)* event handler. However, it is one of the few handlers whose behavior changes between its callback form and its Flash 5-style *onClipEvent( )* form. When implemented as a callback function, *onKeyDown( )* executes only when a key is depressed and the clip specified by `mc` has keyboard focus. By default, keyboard focus is indicated by a yellow rectangle. However, in Flash 6, a couple of caveats make the callback form of *onKeyDown( )* tricky to use. First, a movie clip without button handlers is not focusable unless one or both of its `focusEnabled` or `tabEnabled` properties is explicitly set to `true`. Second, keyboard focus on a movie clip is lost the instant mouse movement over the Flash Player is detected. In the following code, we specify an *onKeyDown( )* callback, then we enable focusing, and finally we focus the clip programmatically. Only then will the clip rotate when keys are pressed.

```
theClip_mc.onKeyDown = function () {
  this._rotation++;
}
theClip_mc.focusEnabled = true;
Selection.setFocus(theClip_mc);
```

If you try this code in Test Movie mode, it will work only when your mouse pointer is not over the movie's Stage; otherwise, focus will immediately be lost, and the event will not fire. These caveats limit the general usefulness of the *onKeyDown( )* movie clip event. Indeed, the *Key.onKeyDown( )* listener event, not *MovieClip.onKeyDown( )*, is Macromedia's intended all-purpose keyboard response tool. However, *MovieClip.onKeyDown( )* proves important when movie clips function as keyboard-controlled interface elements. For example, by implementing *onKeyDown( )* on two separate "menu" movie clips, we can easily use the same shortcut key to trigger different operations, depending on which menu has focus. Unfortunately, this scenario applies strictly to keyboard-only interfaces because even the slightest mouse movement removes the current focus. Macromedia's keyboard-focus implementation is slated for further refinement in future versions of Flash, so a different model may arise to address this situation in Flash 7.

In contrast, the *onClipEvent(keyDown)* version of the handler is intuitive and unchanged from its Flash 5

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

behavior; it executes whenever any key is pressed, period. It requires only that the Flash Player itself be focused. As of Flash 6, the *Key.onKeyDown( )* listener event is usually preferred over *MovieClip.onKeyDown( )*; see complete documentation under *Key.onKeyDown( )*. However, there's one final ingredient in the *onKeyDown( )* stew: the *updateAfterEvent( )* function (which refreshes the screen between frames) works with *MovieClip.onKeyDown( )* but not with *Key.onKeyDown( )*. The general rules of usage, therefore, are:

- If you need a general key-handling mechanism but do not require between-frame screen updates, use *Key.onKeyDown( )*.
- If you need a general key-handling mechanism, but your application must update the screen between frames, use the Flash 5-style *onClipEvent (keyDown)*.
- If you need keyboard control over a movie clip only when it has keyboard focus, use *MovieClip.onKeyDown( )*.

Usage

The *onKeyDown( )* handler does not work with main movie timelines (e.g., `_root`, `_level0`, `_level1`) because main timelines cannot receive input focus.

See Also

Key.onKeyDown( ), *Key.onKeyUp( )*, `MovieClip.focusEnabled`, *MovieClip.onKeyUp( )*, *MovieClip*.`tabEnabled`

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Chapter 18. ActionScript Language Reference

MovieClip.onKeyUp ( ) Event Handler Flash 6

handler executed when a key is released while the clip has focusFlash 5; callback form introduced in

```
mc.onKeyUp()

onClipEvent (keyUp) {
  statements
}
```

Description

The *onKeyUp( )* event handler is the callback form (and more modern analogue) of the legacy *onClipEvent(keyUp)* event handler. However, its behavior changes depending on the form it takes. When implemented as a callback function, *onKeyUp( )* executes only when a key is released while the clip specified by `mc` has keyboard focus. A clip can be focused via the Tab key or *Selection.setFocus( )*, but only when it either defines at least one button handler or when its `tabEnabled` or `focusEnabled` properties are `true`. For example:

```
// Specify the onKeyUp() callback
theClip_mc.onKeyUp = function () {
  this._xscale++;  // Make the clip wider when a key is released.
}
// Allow focusing
theClip_mc.focusEnabled = true;
// Focus the clip
Selection.setFocus(mc);
```

Take heed that keyboard focus on a movie clip is lost the instant mouse movement over the Flash Player is detected. For details, see *MovieClip.onKeyDown( )*.

The *onClipEvent(keyUp)* form of *onKeyUp( )* executes whenever any key is released. It requires only that the Flash Player itself be focused. For example:

```
// Code on a movie clip instance
onClipEvent (keyUp) {
  trace("A key was pressed!");
}
```

As of Flash 6, the *Key.onKeyUp( )* listener event is generally preferred over *MovieClip.onKeyUp( )*; see the *Key* object for much more information.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Usage

In Flash 6, the *updateAfterEvent( )* function (which refreshes the screen between frames) works with *MovieClip.onKeyUp( )*, but not with *Key.onKeyUp( )*.

The *onKeyUp( )* handler does not work with main movie timelines (e.g., `_root`, `_level1`, `_level2`) because main timelines cannot receive input focus.

See Also

Key.onKeyDown( ), *Key.onKeyUp( )*, `MovieClip.focusEnabled`, *MovieClip.onKeyDown( )*, `MovieClip.tabEnabled`

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

MovieClip.onKillFocus ( ) Event Handler Flash 6

handler executed when the clip loses keyboard focus

```
mc.onKillFocus(newFocus)
```

Arguments

newFocus

The *TextField*, *MovieClip*, or *Button* object that now has focus, or `null` if no object has received focus.

Description

The *onKillFocus( )* event handler executes when *`mc`* loses keyboard focus. A movie clip can be keyboard-focused only under one or more of the following circumstances:

- It defines at least one button event handler (*onPress( )*, *onRollOver( )*, etc.).
- Its `tabEnabled` property is `true`, in which case it can be focused by the Tab key.
- Its `focusEnabled` property is `true`, in which case it can be focused by *Selection.setFocus( )*.

A movie clip loses focus in the same manner as a button, as described under *Button.onKillFocus( )*. The *onKillFocus( )* handler can be used to remove a highlight state on a focused interface element. In simple cases, this can also be done with the special `_up` and `_over` frame labels. See *`MovieClip`*`.focusEnabled` and *`MovieClip`*`._focusrect`.

To capture all focus events centrally, rather than for a single instance, use *Selection.onSetFocus( )*.

Usage

Take note that *onKillFocus( )* cannot be used in a Flash 5-style *onClipEvent( )* block.

The *onKillFocus( )* handler does not work with main movie timelines (e.g., `_root`, `_level1`, `_level2`) because main timelines cannot receive input focus.

See Also

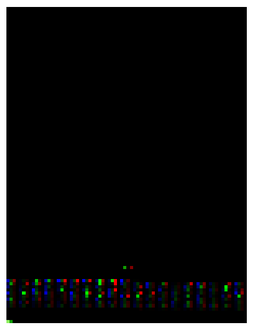
Button.onKillFocus( ), *`MovieClip`*`.focusEnabled`, *`MovieClip`*`.tabEnabled`, *`MovieClip`*`._focusrect`, *Selection.setFocus( )*, *Selection.onSetFocus( )*

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir



Chapter 18. ActionScript Language Reference

MovieClip.onLoad ( ) Event Handler Flash 6

handler executed when the clip first appears on the Stage or when a .swf file finishes loading into the clipFlash 5; callback form introduced in

```
mc.onLoad()

onClipEvent (load) {
  statements
}
```

Description

The *onLoad( )* event handler is the callback form (and more modern analogue) of the legacy *onClipEvent(load)* event handler. The *onLoad( )* event handler executes when a movie clip is born—that is, when a movie clip appears on stage for the first time. It does not apply to main movies (e.g., `_root`, `_level1`, etc.). A movie clip appears on stage (i.e., it triggers an *onLoad( )* event) in one of the following ways:

- The playhead moves onto a keyframe that contains a new instantiation of the clip, placed in the authoring tool.
- The clip is duplicated from another clip via the *duplicateMovieClip( )* or *MovieClip.duplicateMovieClip( )* function.
- The clip is added to the Stage programmatically via the *attachMovie( )* function.
- An external *.swf* file is loaded into the clip with the *loadMovie( )* or *MovieClip.loadMovie( )* function.

The body of an *onLoad( )* event handler is executed *after* any code on the timeline where the movie clip first appears.

When implemented as a Flash 5-style *onClipEvent(load)* block, the *onLoad( )* event handler is often used to initialize variables in a clip or to perform some setup task (such as sizing or positioning a dynamically generated clip). An *onClipEvent(load)* handler can also provide a nice way to prevent a movie clip from automatically playing:

```
// Code placed directly on a movie clip object at authoring time
onClipEvent (load) {
  stop();
}
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

An *onClipEvent(load)* handler can also be used to trigger some function that relies on the existence of a particular clip or loaded *.swf* file in order to execute properly.

The callback form of *onLoad( )* operates quite differently from *onClipEvent(load)*—it has an effect only when assigned at the class level, not the instance level. In the latter case, it will be ignored.

Suppose we assign an *onLoad( )* callback to a clip created with, say, *attachMovie( )* or *duplicateMovieClip( )*:

```
this.attachMovie("BallSymbol", "theBall_mc", 1);
// THIS WILL NEVER EXECUTE:
theBall_mc.onLoad = function () {
  trace("loaded");
};
```

When the first line runs, `theBall_mc` is attached to the current movie clip and the *onLoad( )* phase passes. When the second line runs, a callback function is assigned to `theBall_mc.onLoad`, but it never executes because the clip is already loaded before the callback is assigned. Similarly, an attempt to assign *onLoad( )* from frame 1 of a movie clip will fail:

```
// THIS WILL NEVER EXECUTE:
this.onLoad = function() {
  trace("loaded");
};
```

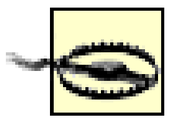
Unlike its Flash 5-style *onClipEvent(load)* cousin, the *onLoad( )* event handler is not applied on a per-instance basis to movie clip instances. Instead, it is used at the class level to initialize the instances of a *MovieClip* subclass. The intended location of an *onLoad( )* callback assignment is, therefore, in a movie clip subclass's `#initclip` block. For example:

```
// CODE ON FRAME 1 OF THE MOVIE CLIP SYMBOL "ballSymbol"
#initclip
// Ball constructor
function Ball () {
  // Initialize the ball instance here...
}

// Set the Ball superclass to MovieClip
Ball.prototype = new MovieClip();

// Associate the Ball class with the Library symbol named "ballSymbol"
Object.registerClass("ballSymbol", Ball);

// Assign the onLoad handler to all instances
Ball.prototype.onLoad = function () {
  trace(this._name + " loaded.");
}
#endinitclip
```

For details on movie clip subclasses, see the `#initclip` pragma and Chapter 14.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir



Assigning *onLoad( )* at the class level, as shown in the *Ball* code, admittedly works, but it is effectively a duplication of roles. Instance initialization code should normally be placed inside the class constructor function.

Note that a clip's *onLoad( )* callback is deleted when an external *.swf* or *.jpg* file is loaded into the clip, but its *onClipEvent(load)* handler is not deleted and will execute across multiple *loadMovie( )* calls. (The *onLoad( )* callback is deleted because an *onLoad( )* event handler property is technically a function stored in a clip's property; clip properties are deleted when a clip is removed from the player.)

Usage

The Flash 5-style *onClipEvent(load)* handler executes when the contents of a clip are unloaded with the *unloadMovie( )* function (because an empty placeholder clip is loaded into the clip when its contents are expelled). In contrast, on its own, an *onLoad( )* callback function will not execute when a clip's contents are unloaded. However, strangely enough, when both types of handlers are applied to the same clip, both will execute when *unloadMovie( )* runs.

Example

The following code uses the *onClipEvent(load)* form of *onLoad( )* to generate an entire field of `star` clips using a single event handler in a cascading chain. The *onClipEvent(load)* handler is attached to a clip instance named `star0`. When the clip loads, it duplicates itself and its *onClipEvent(load)* handler, causing the duplicate, in turn, to duplicate itself. The process stops when the 100th clip is duplicated. Note that we cannot use the callback form of *onLoad( )* to achieve the same effect because callbacks are not copied to duplicated movie clips.

The *.fla* file for this example is available at the online Code Depot:

```
// Attach this onLoad() handler to a clip named star0
onClipEvent (load) {
  // Place the current clip at a random position
  _x = Math.floor(Math.random() * 550);
  _y = Math.floor(Math.random() * 400);

  // Reset clip scale so we don't inherit previous clip's scale
  _xscale = 100;
  _yscale = 100;

  // Randomly size current clip between 50 and 150 percent
  randScale = Math.floor(Math.random() * 100) - 50;
  _xscale += randScale;
  _yscale += randScale;

  // If we're not at the 100th star, make another one
  if (_name != "star100") {
    nextStarNumber = Number(_name.substring(4, _name.length)) + 1;
    this.duplicateMovieClip("star" + nextStarNumber, nextStarNumber);
  }
}
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

MovieClip.onMouseDown( ) Event Handler Flash 6

handler executed when primary mouse button is depressed while the clip is on stageFlash 5; callback form introduced in

```
mc.onMouseDown()

onClipEvent (mouseDown) {
  statements
}
```

Description

The *onMouseDown ( )* event handler is the callback form (and more modern analogue) of the legacy *onClipEvent (mouseDown)* event handler. The *onMouseDown ( )* event handler is identical to the *Mouse.onMouseDown ( )* listener event, except that it allows for screen refreshes between frames via *updateAfterEvent( )* . It executes when `mc` is on stage and the primary mouse button is depressed. The *MovieClip.onMouseDown ( )* handler would be obsolete in Flash 6 if *Mouse.onMouseDown ( )* supported *updateAfterEvent( )* . For complete details, see *Mouse.onMouseDown( )* .

Example

```
// Callback form
_root.onMouseDown = function () {
  trace("The mouse was pressed");
}

// Flash 5-style onClipEvent() block placed directly on a movie clip instanc
onClipEvent (mouseDown) {
  trace("The mouse was pressed");
}
```

See Also

Mouse.onMouseDown( ) , MovieClip.onMouseMove( ) , MovieClip.onMouseUp( )

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

MovieClip.onMouseMove( ) Event Handler Flash 6

handler executed when mouse pointer moves while the clip is on stageFlash 5; callback form introduced in

```
mc.onMouseMove()

onClipEvent (mouseMove) {
  statements
}
```

Description

The *onMouseMove ( )* event handler is the callback form (and more modern analogue) of the legacy *onClipEvent (mouseMove)* event handler. The *onMouseMove ( )* event handler is identical to the *Mouse.onMouseMove ( )* listener event, except that it allows for screen refreshes between frames via *updateAfterEvent( )* . It executes when `mc` is on stage and the mouse pointer moves. The *MovieClip.onMouseMove ( )* handler would be obsolete in Flash 6 if *Mouse.onMouseMove ( )* supported *updateAfterEvent( )* . For complete details, see *Mouse.onMouseMove( )* .

Example

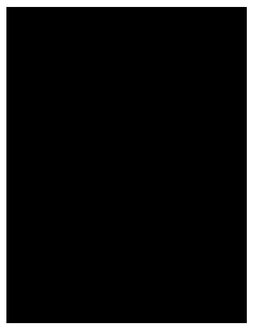
```
// Callback form
_root.onMouseMove = function () {
  trace("The mouse moved");
}

// Flash 5-style onClipEvent() block placed directly on a movie clip instanc
onClipEvent (mouseMove) {
  trace("The mouse moved");
}
```

See Also

Mouse.onMouseMove( ) , MovieClip.onMouseDown( ) , MovieClip.onMouseUp( )

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir



ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

MovieClip.onMouseUp( ) Event Handler Flash 6

handler executed when primary mouse button is released while the clip is on stage

```
mc.onMouseUp()

onClipEvent (mouseUp) {
  statements
}
```

Description

The *onMouseUp ( )* event handler is the callback form (and more modern analogue) of the legacy *onClipEvent (mouseUp* event handler. The *onMouseUp ( )* event handler is identical to the *Mouse.onMouseUp( )* listener event, except that it allows for screen refreshes between frames via *updateAfterEvent ( )* . It executes when *mc* is on Stage and the primary mouse button is released. The *MovieClip.onMouseUp ( )* handler would be obsolete in Flash 6 if *Mouse.onMouseUp ( )* supported *updateAfterEvent( )* . For complete details, see *Mouse.onMouseUp( )* .

Example

```
// Callback form
_root.onMouseUp = function () {
  trace("The mouse was released");
}

// Flash 5-style onClipEvent() block placed directly on a movie clip instanc
onClipEvent (mouseUp) {
  trace("The mouse was released");
}
```

See Also

Mouse.onMouseUp( ) , *MovieClip.onMouseDown( )* , *MovieClip.onMouseMove( )*

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Chapter 18. ActionScript Language Reference

MovieClip.onPress( ) Event Handler Flash 6

Button event; occurs when the mouse is pressed over the clip

```
mc.onPress()

on (press) {
  statements
}
```

Description

The *onPress( )* event handler is identical to the *Button.onPress( )* event, except that it can also be applied to movie clip instances (though not main movies such as `_root`, `_level0`, etc.).

See Also

Button.onPress( ); Section 13.9

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Chapter 18. ActionScript Language Reference

MovieClip.onRelease( ) Event Handler — Flash 6

Button event; occurs when the mouse is pressed and then released over the clip

```
mc.onRelease()

on (release) {
  statements
}
```

Description

The *onRelease( )* event handler is identical to the *Button.onRelease( )* event, except that it can also be applied to movie clip instances (though not main movies such as `_root`, `_level0`, etc.).

See Also

Button.onRelease( ); Section 13.9

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

MovieClip.onReleaseOutside( ) Event Handler Flash 6

Button event; occurs when the mouse is pressed over the clip and then released off the clip

```
mc.onReleaseOutside()

on (releaseOutside) {
  statements
}
```

Description

The *onReleaseOutside( )* event handler is identical to the *Button.onReleaseOutside( )* event, except that it can also be applied to movie clip instances (though not main movies such as `_root`, `_level0`, etc.).

See Also

Button.onReleaseOutside( ); Section 13.9

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

MovieClip.onRollOut( ) Event Handler — Flash 6

Button event; occurs when the mouse pointer moves off of the clip while the mouse button is not depressed

```
mc.onRollOut()

on (rollOut) {
  statements
}
```

Description

The *onRollOut( )* event handler is identical to the *Button.onRollOut( )* event, except that it can also be applied to movie clip instances (though not main movies such as `_root`, `_level0`, etc.).

See Also

Button.onRollOut( ); Section 13.9

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

MovieClip.onRollOver( ) Event Handler — Flash 6

Button event; occurs when the mouse pointer moves over the clip

```
mc.onRollOver()

on (rollOver) {
  statements
}
```

Description

The *onRollOver( )* event handler is identical to the *Button.onRollOver( )* event, except that it can also be applied to movie clip instances (though not main movies such as `_root`, `_level0`, etc.).

See Also

Button.onRollOver( ); Section 13.9

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

MovieClip.onSetFocus( ) Event Handler Flash 6

callback invoked when the clip gains focus

```
mc.onSetFocus(oldFocus)
```

Arguments

oldFocus

The *TextField*, *MovieClip*, or *Button* object that previously had focus, or `null` if no object was previously focused.

Description

The *onSetFocus( )* event handler is executed when *mc* gains focus. A clip gains focus when:

- The user navigates to the clip with the Tab key.
- *Selection.setFocus( )* assigns focus programmatically.

A movie clip can be keyboard-focused under only one or more of the following circumstances:

- It defines at least one button event handler (*onPress( )*, *onRollOver( )*, etc.).
- Its `tabEnabled` property is `true`, in which case it can be focused by the Tab key.
- Its `focusEnabled` property is `true`, in which case it can be focused by *Selection.setFocus( )*.

The following code registers a callback function that responds to *onSetFocus( )*:

```
// Assign the onSetFocus() callback using a function literal
theClip_mc.onSetFocus = function (newFocus) {
  trace(this + " now has focus. Old focus was " + newFocus);
};
```

Notice that from the body of the callback, we can refer to `theClip_mc` using the `this` keyword. To stop a callback from handling the *onSetFocus( )* event, use the *delete* operator, as in:

```
// Make sure to omit the () operator after the function name!
delete clip_mc.onSetFocus;
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

The *onSetFocus( )* handler normally is used to highlight an interface element when it has focus. For example, a dark blue dial can be colored light blue and surrounded with a black circle.

To capture all focus events centrally, rather than for a single instance, use *Selection.onSetFocus( )*.

Usage

Note that *onSetFocus( )* cannot be used in a Flash 5-style *onClipEvent( )* block.

The *onSetFocus( )* handler does not work with main movie timelines (e.g., `_root`, `_level1`, `_level2`) because main timelines cannot receive input focus.

See Also

Button.onSetFocus( ), `MovieClip.focusEnabled`, *MovieClip.onKillFocus( )*, `MovieClip.tabEnabled`, *Selection.onSetFocus( )*, *Selection.setFocus( )*, *TextField.onSetFocus( )*; Chapter 10

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Chapter 18. ActionScript Language Reference

MovieClip.onUnload( ) Event Handler Flash 6

callback invoked when the clip is removed from the Stage or when unloadMovie( ) executesFlash 5; callback form introduced in

```
mc.onUnload()

onClipEvent (unload) {
  statements
}
```

Description

The *onUnload( )* event handler is the callback form (and more modern analogue) of the legacy *onClipEvent (unload)* event handler. The *onUnload( )* event is the opposite of the *onLoad( )* event; it occurs when `mc` expires?that is, it occurs immediately *after* the last frame in which `mc` is present on stage (but *before* the first frame in which it is absent). The *onUnload( )* event does not apply to main movies (e.g., `_root`, `_level1`, etc.)

The following incidents trigger *onUnload( )*:

- The playhead reaches the end of the span of frames upon which `mc` resides.
- `mc` is removed via the *removeMovieClip( )* function (which deletes clips generated by the *attachMovie( )* and *duplicateMovieClip( )* functions).
- The *unloadMovie( )* function is invoked on `mc`.
- `mc` has an external *.swf* file loaded into it.

This last *onUnload( )* event trigger may seem a little odd, but it is actually a natural result of the way movies are loaded into Flash. Anytime a *.swf* or *.jpg* file is loaded into a movie clip, the previous contents of that clip are displaced, triggering an *onUnload( )* event. Here's an example that illustrates the behavior of the *onLoad( )* and *onUnload( )* events in connection with *loadMovie( )*:

1. In the Flash authoring tool, we place an empty movie clip on stage at frame 1 of a movie's main timeline. We name our clip `emptyClip`.
2. At frame 5 of the main timeline, we load the movie test.swf into `emptyClip` using the following code:

```
emptyClip.loadMovie("test.swf");
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

3. We play the movie using Control → Play movie.

The results are:

1. The `emptyClip` clip appears on frame 1, causing an *onLoad( )* event.
2. On frame 5, the *loadMovie( )* function is executed in two stages:
    a. The placeholder content of `emptyClip` is removed to make room for the incoming test.swf, causing an *onUnload( )* event.
    b. The movie test.swf loads, causing an *onLoad( )* event.

The *onUnload( )* event typically is used to perform housecleaning code?code that cleans up the Stage or resets the program environment in some way. An *onUnload( )* handler also provides a means for performing some action (such as playing another movie) after a movie clip ends.

Unlike its cousin *onLoad( )*, whose callback form has limited use, *onUnload( )* works identically whether defined as a callback or in an *onClipEvent(unLoad)* block. However, the usual differences between callback handlers and *onClipEvent( )* handlers still apply:

- When the contents of *`mc`* are unloaded, the *onUnload( )* callback is lost, but the *onClipEvent(unload)* handler is retained.
- When *`mc`* is duplicated via *duplicateMovieClip( )*, the *onUnload( )* callback is not copied to the duplicate clip, but the *onClipEvent(unLoad)* handler is.

Usage

Perhaps surprisingly, in Flash 6, *onUnload( )* is not triggered when the Flash Player is closed or when `_level0` (the main movie) is removed from the Player. Macromedia may address this idiosyncrasy in the future.

See Also

MovieClip.onLoad( ), *MovieClip.unloadMovie( )*

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Chapter 18. ActionScript Language Reference

MovieClip._parent Property — Flash 4

a reference to the host clip or movie containing this clip — ***read-only***

mc._parent

Description

The _parent property stores a reference to the clip object upon whose timeline *mc* resides. Main movies don't support the _parent property because they are at the top level (conveniently referred to as _root). References to _root._parent return undefined. The _parent property gives us the powerful ability to manipulate clips in relation to the current clip.

Bugs

Though it is possible in Flash 5 and Flash 6 to reassign the _parent property to another clip, doing so has little purpose?only the reference is changed, not the physical structure of the clips. This unintentional behavior will likely be changed in the future.

Example

If *mc* resides on the main timeline of a movie, we can play the main timeline from *mc* using:

```
_parent.play();
```

We can also set the parent timeline's properties, as in:

```
_parent._alpha = 50;
```

The _parent property can also be used in succession; that is, we can access the _parent of a _parent, as follows:

```
_parent._parent.play();  // Play the clip two clips above the current clip
```

See Also

_root; Section 13.5.3

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Chapter 18. ActionScript Language Reference

MovieClip.play( ) Method Flash 5

begin the sequential display of frames in a clip or movie

```
mc.play()
```

Description

The *MovieClip.play( )* method is an alternative to the global *play( )* function. Use the *MovieClip* method form to control remote movie clips or main movies specified by `mc`.

For general usage instructions, see the global *play( )* function.

Example

```
// Begin playing clip named intro_mc
intro_mc.play();
```

See Also

MovieClip.stop( ), play( )

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Chapter 18. ActionScript Language Reference

MovieClip.prevFrame( ) Method Flash 5

send a clip or movie's playhead backward one frame and stop it there

```
mc.prevFrame()
```

Description

The *MovieClip.prevFrame( )* method is an alternative to the global *prevFrame( )* function. Use the *MovieClip* method form to control remote movie clips or main movies specified by *mc*.

For general usage instructions, see the global *prevFrame( )* function.

Example

```
// Rewind slideshow_mc one frame and stop it
slideshow_mc.prevFrame();
```

See Also

MovieClip.nextFrame( ), prevFrame( )

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

MovieClip.removeMovieClip( ) Method Flash 5

delete a movie clip from the Flash Player

```
mc.removeMovieClip()
```

Description

The *MovieClip.removeMovieClip( )* method is an alternative to the global *removeMovieClip( )* function. When invoked as a *MovieClip* method, *removeMovieClip( )* takes no `target` parameter; it always removes `mc`. The *MovieClip* method syntax is less prone to user error than its global function counterpart.

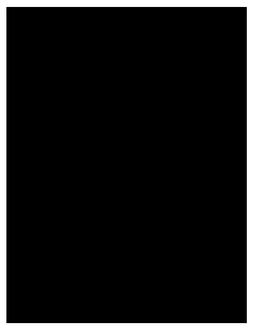
For usage instructions, see the global *removeMovieClip( )* function.

See Also

duplicateMovieClip( ), *MovieClip.attachMovie( )*, *MovieClip.duplicateMovieClip*( ), *removeMovieClip( )*; Section 13.7

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir



Chapter 18. ActionScript Language Reference

MovieClip._rotation Property — Flash 4

rotation, in degrees, of a clip or movie — ***read/write***

```
mc._rotation
```

Description

The floating-point `_rotation` property specifies the number of degrees *`mc`* is rotated from its original orientation (if *`mc`* is not a main movie, the *original orientation* is that of *`mc`*'s symbol in the Library). Both authoring-time and programmatic adjustments are reflected in `_rotation`. Numbers in the range of 0 to 180.0 rotate the clip clockwise. Numbers in the range of 0 to -180.0 rotate the clip counterclockwise. The same effect is achieved when rotating a clip *`n`* degrees or *`n-360`* degrees (where *`n`* is positive). For example, there is no difference between rotating a clip +299.4 degrees or -60.6 degrees. Likewise, when *`n`* is negative, there is no difference between *`n`* degrees and *`n+360`* degrees. For example, rotating a clip -90 degrees is the same as rotating it +270 degrees.

When `_rotation` is set to anything outside the range of -180 to 180, the value is brought into the proper range according to the equivalent of the following calculation:

```
x = newRotation % 360;
if (x > 180) {
  x -= 360;
} else if (x < -180) {
  x += 360;
}
_rotation = x;
```

Bugs

In Flash Player 4, setting the `_rotation` of a clip reduces the scale of that clip by a fractional amount. Over many `_rotation` settings, a clip will actually shrink noticeably. To account for this bug, set the `_xscale` and `_yscale` of the clip to 100 (or whatever their previous values are) whenever setting the `_rotation`.

Example

The following *onEnterFrame( )* handler causes the `box_mc` clip to rotate 5 degrees clockwise each time a frame is rendered:

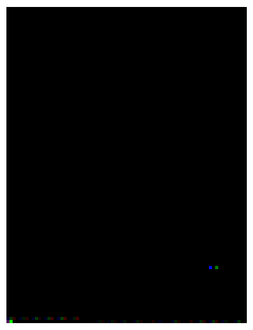
```
box_mc.onEnterFrame = function {
  this._rotation += 5;
}
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir



Chapter 18. ActionScript Language Reference

MovieClip.setMask( ) Method — Flash 6

assigns a movie clip as a mask for another clip

```
mc.setMask(maskMovieClip)
```

Arguments

maskMovieClip

A string indicating the path to the movie clip to be used as `mc`'s mask. Because movie clip references are converted to paths when used in a string context, `maskMovieClip` can also be a movie clip object reference, as in `mc.setMask(maskClip)` versus `mc.setMask("maskClip")`.

Description

The *setMask( )* method assigns `maskMovieClip` as a mask for `mc`. When invoked, the content of `maskMovieClip` becomes a hole through which the content of `mc` is visible. Those portions of `mc` beneath occupied pixels of `maskMovieClip` are displayed, but portions of `mc` beneath empty pixels in `maskMovieClip` are hidden. Therefore, masks typically are created with black pixels in the areas where you want the masked clip, `mc`, to show through. For example, `maskMovieClip` may contain a filled black circle used as a spotlight on `mc`'s content. `To` remove any mask currently applied to `mc`, call *setMask( )* with a `maskMovieClip` parameter of `null`.

The following code sets `flashlight_mc` as the mask of `darkroom_mc`:

```
darkroom_mc.setMask(flashlight_mc);
```

And the following code removes the `flashlight_mc` mask:

```
darkroom_mc.setMask(null);
```

Shapes drawn with the movie clip drawing methods (*beginFill( )*, *lineTo( )*, *curveTo( )*, etc.) can be used in masks. As of Flash 6 (but not before), timelines of movie clips used as masks play normally. Each frame's content becomes the mask when the content is displayed. Mask clips can also be dragged, sized, and moved programmatically, and they can take movie clip and button-style event handlers. For example, a mask clip can define an *onEnterFrame( )* handler that changes its position with each passing frame. Or, a mask clip can define *onPress( )* and *onRelease( )* handlers that drag and drop it, as follows:

```
darkroom_mc.setMask("spotlight_mc");

spotlight_mc.onPress = function () {
  trace("Dragging spotlight");
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

```
  this.startDrag();
}

spotlight_mc.onRelease = spotlight_mc.onReleaseOutside = function () {
  trace("Dropping spotlight");
  this.stopDrag();
}
```

When *maskMovieClip* defines at least one button event handler or contains a button object, no button events pass through to *mc*. That is, a masked clip cannot respond to button events when its mask defines a button event or contains a button.

If we apply a runtime mask to a movie clip used in an author-time mask layer, Flash removes the author-time mask and places the newly masked clip behind the layers it masked at authoring time.

Usage

Masks in Flash have several limitations:

- Prior to Flash Player 6.0.40.0, text fields without embedded fonts cannot be masked with a mask movie clip. As of Flash Player 6.0.40.0, the rectangular bounding box of a runtime mask movie clip can mask a text field even when its fonts are not embedded. To mask a text field in older versions of the Player, be sure to embed its font(s).
- Text fields without embedded fonts cannot be used as masks.
- Text fields without embedded fonts cannot be masked with an author-time mask layer.
- Lines cannot be used as masks. Use fills, not lines, in mask movie clips. Lines can be turned into fills via Modify Shape Convert Lines to Fills.
- The *setMask( )* method cannot assign more than one clip to be masked by a single mask movie clip. To apply an identical mask to several clips, duplicate the original mask movie clip as often as needed, or use an author-time mask, which can be applied to multiple movie clips simultaneously.
- In Flash 6, fully contained holes in shapes inside movie clips are not reflected by masks. For example, it is impossible to create a doughnut-shaped mask using a movie clip (the doughnut's hole will be filled, and the mask will simply be a solid circle). However, an author-time primitive shape in a mask layer can be used to create mask shapes with fully contained holes. The *beginFill( )* method can also be used at runtime to draw a shape with a hole that functions properly as a mask.

See Also

MovieClip.beginFill( )

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Chapter 18. ActionScript Language Reference

MovieClip.startDrag( ) Method Flash 5

make a movie or movie clip follow the mouse

```
mc.startDrag()
mc.startDrag(lockCenter)
mc.startDrag(lockCenter, left, top, right, bottom )
```

Arguments

lockCenter

A Boolean indicating whether the registration point of `mc` should be centered under the mouse pointer (`true`) or dragged relative to its original location (`false`).

left

A number specifying the x-coordinate to the left of which `mc`'s registration point cannot be dragged.

top

A number specifying the y-coordinate above which `mc`'s registration point may not be dragged.

right

A number specifying the x-coordinate to the right of which `mc`'s registration point cannot be dragged.

bottom

A number specifying the y-coordinate below which `mc`'s registration point cannot be dragged.

Description

The *MovieClip.startDrag( )* method is an alternative to the global *startDrag( )* function. When invoked as a *MovieClip* method, *startDrag( )* does not take a `target` parameter; it drags `mc`. The *MovieClip* method syntax is less prone to user error than its global function counterpart because no `target` parameter is required.

For usage instructions, see the global *startDrag( )* function.

Bugs

Note that the correct order of the constraining rectangular coordinates is left, top, right, bottom, but the Flash 5 ActionScript Dictionary lists them in the wrong order under *MovieClip.startDrag( )*.

Example

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

```
// Code to drag and drop the current clip or movie
this.onPress = function () {
  this.startDrag(true);
};

this.onRelease = function () {
  stopDrag();
}
```

See Also

MovieClip.stopDrag( ), *startDrag( )*, *stopDrag( )*; Section 13.7

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Chapter 18. ActionScript Language Reference

MovieClip.stop( ) Method Flash 5

pause a clip or movie

```
mc.stop()
```

Description

The *MovieClip.stop( )* method is an alternative to the global *stop( )* function. Use the *MovieClip* method form to control remote movie clips or main movies specified by *mc*. The Flash Player continues to render the Stage's contents and process events even when the playhead is stopped.

For general usage instructions, see the global *stop( )* function.

Example

```
// Halt the playback of spinner_mc
spinner_mc.stop();
```

See Also

MovieClip.nextFrame( ), MovieClip.play( ), MovieClip.prevFrame( ), stop( )

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Chapter 18. ActionScript Language Reference

MovieClip.stopDrag( ) Method Flash 5

end a drag operation in progress

```
mc.stopDrag()
```

Description

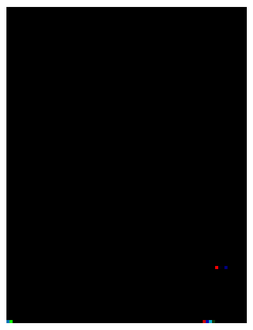
The *MovieClip.stopDrag( )* method is an alternative to the global *stopDrag( )* function. However, there is no need to use the *MovieClip* method form; *stopDrag( )* cancels any drag operation in progress, whether invoked through a movie clip or as a global function.

See Also

MovieClip.startDrag( ), stopDrag( )

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir



Chapter 18. ActionScript Language Reference

MovieClip.swapDepths( ) Method Flash 5

change the graphic layering of an instance in the visual content stack

```
mc.swapDepths(target)
mc.swapDepths(depth)
```

Arguments

target

A string indicating the path to the movie clip to be swapped with `mc`. Because movie clip references are converted to paths when used in a string context, `target` can also be a movie clip object reference, as in *mc.swapDepths(*`window2`*)* versus *mc.swapDepths(*`"window2"`*)*.

depth

An integer, between -16384 and 1048575 (inclusive), specifying a level in `mc`'s parent's clip stack. Higher depths obscure lower depths, so a depth of 6 is in front of 5, which is in front of 4, and so on. Depths above -1 are reserved for dynamically generated content; depths from -16383 to -1 (inclusive) are reserved for author-time content; and depth -16384 is reserved for dynamic content that appears beneath all author-time content in `mc` (e.g., a runtime background). For important details, see Section 13.4.

Description

All movie clip instances in a *.swf* document reside in a stack that governs the visual layering of instances in the Player. The stack is structured like a deck of cards; clips in higher positions in the stack appear in front of clips in lower positions. The position of an instance in the stack is initialized when the clip is created (either in the Flash authoring tool or via *attachMovie( )*, *duplicateMovieClip( )*, or *createEmptyMovieClip( )*). Using *swapDepths( )*, we can change the position of an instance in the stack.

The *swapDepths( )* method takes two forms. When used with a `target` argument, *swapDepths( )* trades the stack position of `mc` and `target`, provided that `mc` and `target` share the same parent clip (i.e., they reside on the same timeline). When used with a `depth` argument, *swapDepths( )* places `mc` in a new position in `mc`'s parent stack. If that position is occupied, the previous occupant is moved to `mc`'s old position. Note that *swapDepths( )* cannot be used to move a clip outside of the stack maintained by its parent. For information on how movies and movie clips are stacked in the Player, see Section 13.4. To determine the current depth of a clip, use *MovieClip.getDepth( )*.

Bugs

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Note that swapping a duplicated or attached instance with a manually created instance via the `depth` argument can cause redraw problems in Flash Player 5. Treat the `depth` argument with prudence, and always test depth-swapping code extensively.

See Also

MovieClip.attachMovie( ), *MovieClip.createEmptyMovieClip( )*,, *MovieClip.createTextField( )*, *MovieClip.duplicateMovieClip*( ), *MovieClip.getDepth( )*; Section 13.4

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Chapter 18. ActionScript Language Reference

MovieClip.tabChildren Property — Flash 6

controls whether contained objects are included in the automatic tab order — ***read/write***

mc.tabChildren

Description

The Boolean `tabChildren` property specifies whether the text fields and buttons inside *mc* are included in the automatic tab order. It applies only to movie clips, not main movies (e.g., `_root`, `_level1`). When `tabChildren` is `true` (the default), all buttons, movie clips with button handlers, and text fields contained by *mc* are part of the automatic tab order, even when `mc.tabEnabled` is `false`. When `tabChildren` is `false`, the objects contained by *mc* are not accessible via the Tab key.

The `tabChildren` property has two typical uses:

- It can conveniently disable tabbing for a whole group of objects. For example, to disable a set of toolbar buttons, place the buttons inside a movie clip and set the clip's `tabChildren` to `false`.
- It can prevent a movie clip's contents from interfering with the clip's keyboard accessibility. For example, to prevent the Tab key from accessing, say, the handle inside a slider bar, set the slider bar's `tabEnabled` to `true` and `tabChildren` to `false`. This way, the entire bar counts as a single tab index in the tab order.

Usage

The Tab key does not work reliably in Test Movie mode unless Control Disable Keyboard Shortcuts is enabled. Be sure to test keyboard behavior in the Standalone Player, if applicable, and in the Flash Player for all targeted web browsers.

Example

The following code enables `sliderBar_mc` to receive focus via the Tab key, but it prevents any of its contents from being focused:

```
sliderBar_mc.tabEnabled = true;
sliderBar_mc.tabChildren = false;
```

See Also

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

MovieClip.tabEnabled, *MovieClip*.tabIndex

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Chapter 18. ActionScript Language Reference

MovieClip.tabEnabled Property — Flash 6

includes or excludes the movie clip from the current tab order — ***read/write***

mc.tabEnabled

Description

The Boolean `tabEnabled` property governs whether *mc* is part of the *tab order* for the text fields, movie clips, and buttons currently on screen. The tab order is the sequence in which objects are focused when the Tab key is pressed in a movie. See *TextField* .tabEnabled for complete details.

For movie clips, the `tabEnabled` property can:

- Exclude a movie clip with a defined `tabIndex` from the custom tab order
- Exclude any clip with button event(s) from the automatic tab order
- Include a movie clip in the automatic tab order (the automatic tab order does not, by default, include clips without button events)

By default, `tabEnabled` is `undefined` and *mc* is excluded from the automatic tab order unless it defines at least o button handler (*onPress ( )* , *onRelease ( )* , etc.), in which case it is included in the automatic tab order.

A movie clip that does not define button handlers typically is added to the automatic or custom tab order when it is an interface element that is keyboard accessible. For example, a listbox movie clip should be focusable via the Tab key and controllable via the arrow keys, so it should have a `tabEnabled` property of `true` .

By default, when a movie clip has focus, Flash surrounds it with a yellow rectangle. To supply a custom highlight state, w should first turn the yellow rectangle off by setting *MovieClip* ._focusrect to `false` , and then either create th highlight state inside the clip's *onSetFocus ( )* handler, or define `_up` and `_over` frames in the clip that correspond to the unfocused and focused states, respectively. (Flash automatically displays those special frames when the clip loses and gai focus.)

To allow a movie clip to be focused programmatically via *Selection.setFocus ( )* , we must set its `focusEnabled` prop to `true` . Note that the `tabEnabled` property applies to the Tab key only, while the `focusEnabled` property appli to *Selection.setFocus ( )* only. For many interface widgets, it is appropriate to disable automatic tabbing for all of the clip's *children* (the nested clips, buttons, and text fields it contains). We can do this conveniently by setting the `tabChildre` property to `false` . Disabling automatic child tabbing allows us to dictate the keyboard behavior of a clip explicitly, wit worrying about interference from any subordinate interface elements it may contain.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

In general, to enable focus for a movie clip interface widget we combine the following property settings:

```
// Disable automatic yellow focus rectangle
theClip_mc._focusrect = false;
// Allow the clip to be focused by the Tab key
theClip_mc.tabEnabled = true;
// Allow the clip to be focused by Selection.setFocus()
theClip_mc.focusEnabled = true;
// Prevent automatic tabbing to clips, text fields, and buttons inside the c
theClip_mc.tabChildren = false;
```

Note that when a clip's `tabEnabled` is `false` , any text fields, buttons, or movie clips it contains may still have tabbi enabled. To disable tabbing for a clip and all its contained objects, set both `tabEnabled` and `tabChildren` to `fal` .

Movie clips that behave as buttons (i.e., that define button event handlers) normally have `tabEnabled` set to `false` v they are disabled (when the `enabled` property is set to `false`) or when they are accessible only to mouse input, such games that use transparent buttons to detect mouse hit regions.

Usage

The Tab key does not work reliably in Test Movie mode unless Control Disable Keyboard Shortcuts is enabled. Be s to test keyboard behavior in the Standalone Player, if applicable, and in the Flash Player for all targeted web browsers.

See Also

`MovieClip` .enabled , *`MovieClip` .focusEnabled* , *MovieClip.onSetFocus( )* , *`MovieClip` .tabChildren* , *`MovieClip` .tabIndex* , *Selection.setFocus( )* , *`TextField` .tabEnabled*

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Chapter 18. ActionScript Language Reference

MovieClip.tabIndex Property — Flash 6

specifies the movie clip's index in the custom tab order — ***read/write***

mc.tabIndex

Description

The integer `tabIndex` property specifies *mc*'s position in the custom tab order. When no text field, clip, or button has a `tabIndex` property defined, Flash determines tab order automatically, and pressing the Tab key moves focus between movie clips with button handlers, buttons, and text fields from right to left, then top to bottom. We use the `tabIndex` property to force Tab key navigation to follow our own sequence, overriding the automatic order. For complete details, see *`TextField`*`.tabIndex`.

Objects with `_visible` set to `false` are not included in the tab order. When at least one object has `tabIndex` set, any objects without an assigned `tabIndex` cannot be accessed via the Tab key.

By default, movie clips are not tab-enabled unless they define button handlers. To include in the automatic tab order a movie clip that does not define button handlers, set its `tabEnabled` property to `true`. To include any movie clip in a custom tab order, set its `tabIndex` property to an integer (the `tabEnabled` property need not also be set). A movie clip without button handlers typically is added to the automatic or custom tab order when it is an interface element that is keyboard accessible (in which case, it typically responds to keyboard control via either *Key.onKeyUp( )* or *MovieClip.onKeyUp( )*).

Usage

The Tab key does not work reliably in Test Movie mode unless Control Disable Keyboard Shortcuts is enabled. Be sure to test keyboard behavior in the Standalone Player, if applicable, and in the Flash Player for all targeted web browsers.

Example

The following code restricts Tab key access to a listbox movie clip, a text field, and a Submit button:

```
phoneListBox_mc.tabIndex = 1;
firstName_txt.tabIndex = 2;
submit_btn.tabIndex = 3;
```

See Also

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

MovieClip.tabChildren, *MovieClip*.tabEnabled, *TextField*.tabIndex

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

MovieClip._target Property Flash 4

the target path of a clip or movie, in slash syntax ***read-only***

```
mc._target
```

Description

The `_target` property represents the path to *mc* in a Flash 4-style slash notation string. For example, if a clip, `ball`, resides on the main movie timeline, `ball`'s `_target` property is "`/ball`". A clip named `stripe` inside `ball` would have a `_target` of "`/ball/stripe`".

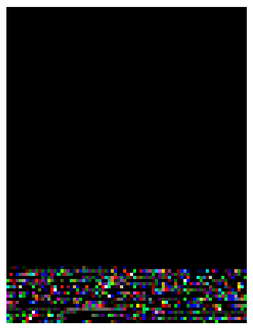
To retrieve a string giving the path to a clip in dot notation, use the global *targetPath( )* function. To convert the slash notation string into an object reference, use *eval( )*.

See Also

eval( ), targetPath( )

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir



Chapter 18. ActionScript Language Reference

MovieClip._totalframes Property — Flash 4

the total number of frames in a clip or movie timeline — ***read-only***

`mc._totalframes`

Description

The integer `_totalframes` property represents the number of frames in `mc`'s timeline, where `mc` is a movie clip or main movie. The `_totalframes` property is most often used along with `_framesloaded` to create preloaders, as shown in the entry for `MovieClip._framesloaded`.

The *loadMovie( )* and *unloadMovie( )* functions are executed asynchronously. Hence, the value of `_framesloaded` varies according to when it is called and the value of `mc`, as follows:

- If `mc` is `undefined`, `_totalframes` is `undefined`.
- After a *loadMovie( )* or *unloadMovie( )* statement occurs in source code, but before it executes, and before the number of frames of the external file has been determined, `_totalframes` returns the existing number of frames in `mc` (prior to the *loadMovie( )* or *unloadMovie( )* statement).
- After the number of frames is determined, `_totalframes` returns the number of frames in the loading movie, unless the file at `url` is not found.
- If the file at `url` is not found and `mc` is a movie clip, `_totalframes` returns 0 and `_framesloaded` returns -1. However, if `mc` is a document level, such as `_level0`, both `_framesloaded` and `_totalframes` return `undefined`.
- After an *unloadMovie( )* command executes, `_totalframes` returns 0 if `mc` is a movie clip and `undefined` if `mc` is a document level, such as `_level0`.

See Also

`MovieClip._framesloaded`

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Chapter 18. ActionScript Language Reference

MovieClip.trackAsMenu Property — Flash 6

modifies the onRelease( ) handler requirements, enabling menu-style behavior — ***read/write***

mc.trackAsMenu

Description

The Boolean `trackAsMenu` property modifies the conditions that cause *mc*'s *onRelease( )* event handler to execute. It normally is used to turn a group of buttons into a menu and applies only to movie clips that define at least one button event handler. For complete details, see *Button*.`trackAsMenu`.

See Also

Button.`trackAsMenu`, *MovieClip.onRelease( )*

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Chapter 18. ActionScript Language Reference

MovieClip.unloadMovie( ) Method Flash 5

remove a movie or movie clip from the Player

```
mc.unloadMovie()
```

Description

The *MovieClip.unloadMovie( )* method is an alternative to the global *unloadMovie( )* function. When invoked as a *MovieClip* method, *unloadMovie( )* does not take a `target` parameter; it always unloads `mc`. The *MovieClip* method syntax is less prone to user error than its global function counterpart because no `target` parameter is required.

For usage instructions, see the global *unloadMovie( )* function.

Example

```
// Removes a loaded document from level 1
_level1.unloadMovie();
```

See Also

MovieClip.loadMovie( ), *unloadMovie( )*; Section 13.7

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Chapter 18. ActionScript Language Reference

MovieClip._url Property — Flash 4

the network address from which a clip or movie was loaded — ***read-only***

mc._url

Description

The string _url property represents the URL (Uniform Resource Locator) indicating the Internet or local disk location from which the content of *mc* was loaded. The _url property is always an absolute URL, never a relative one. For main movies, _url is simply the location of the current *.swf* file. The _url of all movie clips in a *.swf* file is the same as the main movie of that file unless external *.swf* files have been loaded into individual clips via *MovieClip.loadMovie( )*. The _url of a clip that hosts an externally loaded *.swf* file is the location of the externally loaded file.

The _url property sometimes is used to create simple security systems that prevent a movie from playing when displayed in a unwanted location.

Example

The value of _url in a movie loaded from a web site looks like this:

```
"http://www.moock.org/gwen/meetgwen.swf"
```

The value of _url in a movie loaded from a local PC hard drive looks like this:

```
"file:///C|/data/flashfiles/movie.swf"
```

Here we check whether a movie is hosted in the desired location (if it's not, we display a frame that contains an error):

```
if (this._url != "http://www.moock.org/gwen/meetgwen.swf") {
  trace("This movie is not running from its intended location.";
  this.gotoAndStop("accessDenied");
}
```

See Also

MovieClip.loadMovie( ), *System.security.allowDomain( )*

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Chapter 18. ActionScript Language Reference

MovieClip.useHandCursor Property — Flash 6

dictates whether a hand pointer is displayed when the mouse is over the clip (button-enabled clips only) — *read/write*

```
mc.useHandCursor
```

Description

When set to `false`, the Boolean `useHandCursor` property disables the display of the hand pointer normally shown when the mouse is over *`mc`*'s hit area. It applies only to movie clips with button event handlers. For details, see *`Button`*`.useHandCursor`.

Example

The following code disables the hand cursor for `checkBox_mc`:

```
checkBox_mc.useHandCursor = false;
```

The following code disables the hand cursor for all movie clips in a movie:

```
MovieClip.prototype.useHandCursor = false;
```

See Also

`Button``.useHandCursor`, *Mouse.hide*( )

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Chapter 18. ActionScript Language Reference

MovieClip.valueOf( ) Method Flash 5

a reference to a clip or movie

```
mc.valueOf()
```

Returns

An object reference to *mc*.

Description

The *valueOf( )* method returns an object reference to *mc*. The reference is synonymous with *mc* itself, so it is rarely, if ever, necessary to invoke *valueOf( )* explicitly.

See Also

Object.valueOf( ), *targetPath( )*

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Chapter 18. ActionScript Language Reference

MovieClip._visible Property — Flash 4

whether a clip or movie is shown or hidden — ***read/write***

mc._visible

Description

The Boolean _visible property indicates whether *mc* is currently shown (true) or hidden (false). We use _visible as a quick means of hiding a movie clip or movie from view. Note that button events and the *onKeyUp( )* and *onKeyDown( )* event handler properties do not fire when a clip is not visible, but otherwise the clip operates as normal. It can still be controlled via ActionScript; it can still be played, stopped, and detected with *hitTest( )*; and it can still receive other movie clip events. Invisible clips are simply not displayed on screen.

The initial value of the _visible property for all movie clips is true, even for movie clips that are fully transparent or completely off stage. The _visible property changes only when it is deliberately modified by a script; think of it as a way to show and hide clips programmatically, not as a reflection of all factors (such as position and transparency) that can affect a clip's visibility.

Hiding a movie clip using the _visible property is preferred to setting it to fully transparent or moving it off stage, because the Flash Player does not attempt to draw the graphics in movie clips with _visible set to false, thus improving rendering performance.

Example

The following event handlers hide the current clip when the mouse button is pressed and reveal it when the mouse button is released:

```
this.onMouseDown = function () {
  this._visible = false;
}

this.onMouseUp = function () {
  this._visible = true;
}
```

See Also

MovieClip._alpha

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Chapter 18. ActionScript Language Reference

MovieClip._width Property Flash 5

width of a clip or movie, in pixelsFlash 4; enhanced in Flash 5read-only in Flash 4; read/write as of

mc._width

Description

The floating-point _width property stores a nonnegative number specifying the current width of *mc* , in pixels. If *mc* no content, _width is 0. The _width property measures the content of a clip or movie as the distance between the leftmost occupied pixel and the rightmost occupied pixel, even if there is empty space between those pixels. An *occupied pixel* is a pixel that contains a shape, graphic, button, movie clip, text field, or other content element. Changes made to a clip's width in the authoring tool or via _xscale are reflected by _width .

Setting the value of _width resizes a movie clip horizontally. Attempts to set _width to a negative value are ignored. Setting a clip's _width to 0 does not hide the clip; it turns it into a one-pixel vertical line. Flash's basic unit of measure i a twentieth of a pixel (a "twip"), so the shortest legal _width change is .05 pixels. Smaller increments are ignored:

```
trace(ball_mc._width);    // Displays: 100
ball_mc._width += .01;
trace(ball_mc._width);    // Still displays 100. The assignment had no effec
```

The _width of the main movie (either _root._width or _level*n* ._width) is not the Stage width as specifi under Modify Document Dimensions (or Modify Movie Dimensions in Flash 5) in the authoring tool, but rather the width of the contents of the main movie. As of Flash 6, the movie's specified dimensions are available via Stage.height and Stage.width .

For a loading movie, the _width property returns undefined until the movie has completely loaded.

Usage

Note that when we set the width of a movie clip, lines are scaled proportionally, losing their original point size as set in th Stroke panel. However, the point size of lines set to Hairline in the Stroke panel is not scaled when a movie clip is resized Use hairlines to prevent your strokes from appearing fat or distorted when a clip is scaled.

Example

```
ball_mc._width = 20;          // Set the width of ball_mc to 20 pixels
ball_mc._width *= 2;          // Double the width of ball_mc
```

See Also

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

MovieClip ._height,*MovieClip* ._xscale,Stage.width

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Chapter 18. ActionScript Language Reference

MovieClip._x Property — Flash 4

horizontal location of a clip or movie, in pixels — *read/write*

mc._x

Description

The floating-point _x property always indicates the horizontal position of *mc*'s *registration point*, but it is measured relative to one of three possible coordinate spaces:

- If *mc* resides on the main timeline, _x is measured relative to the *Stage's left edge*. For example, an _x of 20 indicates that the registration point of *mc* is 20 pixels to the right of the Stage's left edge; -20 indicates 20 pixels to the left.
- If *mc* resides on another movie clip instance's timeline, _x is measured relative to the *registration point* of that parent instance. For example, an _x of 20 indicates that the registration point of *mc* is 20 pixels to the right of its parent instance's registration point; -20 indicates 20 pixels to the left.
- If *mc* is the main movie, _x is the horizontal offset of the entire *.swf* document, relative to the Stage's left edge. For example, an _x of 200 indicates that the contents of the Stage are offset 200 pixels to the right of their author-time position; -200 indicates 200 pixels to the left.

The _x property (along with all horizontal coordinates in Flash) increases to the right and decreases to the left. Fractional _x values are approximated in Flash with antialiasing (blurring). Flash's basic unit of measure is a twentieth of a pixel (a "twip"), so the shortest distance a clip can be moved horizontally is .05 pixels. Smaller increments are ignored:

```
trace(ball_mc._x);    // Displays: 0
ball_mc._x += .01;
trace(ball_mc._x);    // Still displays 0. The assignment had no effect.
```

If *mc* is contained by an instance that is scaled and/or rotated, the coordinate space it inhabits is also scaled and/or rotated. For example, if *mc*'s parent is scaled by 200% and rotated 90 degrees clockwise, _x will increase in the downward direction rather than to the right, and a single unit of _x will actually be 2 pixels instead of 1.

Because switching between instance and main movie coordinate spaces can be cumbersome, the *MovieClip* object provides the *localToGlobal( )* and *globalToLocal( )* methods for performing coordinate-space transformations.

Example

The following *onEnterFrame( )* handler causes ball_mc to move 5 pixels to the right with each passing frame

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

(assuming that its coordinate space hasn't been altered by transformations to its parent):

```
ball_mc.onEnterFrame = function () {
  this._x += 5;
}
```

Positioning clips via `_x` and `_y` is a fundamental task in visual ActionScript programming. The following example adds to all movie clips a *follow( )* method that moves a clip towards a point at a fixed rate specified by the custom `speed` property (many other motion samples can be obtained from the online Code Depot):

```
// Method: MovieClip.follow()
// Moves a clip toward the point (leaderX, leaderY)
MovieClip.prototype.follow = function (leaderX, leaderY) {
  // Move only if we're not at the destination
  if (this._x != leaderX || this._y != leaderY) {
    // Determine the distance between the clip and leader
    var deltaX = this._x - leaderX;
    var deltaY = this._y - leaderY;
    var dist = Math.sqrt((deltaX * deltaX) + (deltaY * deltaY));

    // Allocate speed between X and Y axes
    var moveX = this.speed * (deltaX / dist);
    var moveY = this.speed * (deltaY / dist);

    // If the clip has enough speed to overshoot the destination, just go
    // to the destination. Otherwise move according to the clip's speed.
    if (this.speed >= dist) {
      this._x = leaderX;
      this._y = leaderY;
    } else {
      this._x -= moveX;
      this._y -= moveY;
    }
  }
}

// Usage...
butterfly_mc.speed = 10;
butterfly_mc.onEnterFrame = function () {
  this.follow(_root._xmouse, _root._ymouse);
}
```

See Also

MovieClip.globalToLocal( ), *MovieClip.localToGlobal( )*, *`MovieClip`*`._xscale`, *`MovieClip`*`._y`

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Chapter 18. ActionScript Language Reference

MovieClip._xmouse Property — Flash 5

horizontal location of the mouse pointer — ***read-only***

mc._xmouse

Description

The floating-point _xmouse property indicates the horizontal location of the mouse pointer's hotspot, relative to the coordinate space of *mc*. If *mc* is a main movie, _xmouse is measured from the Stage's left edge. If *mc* is an instance, _xmouse is measured from the instance's registration point. To obtain a consistent _xmouse coordinate that is always measured relative to the Stage, use _root._xmouse.

Example

Placing the following code on a clip causes it to follow the mouse pointer's horizontal position relative to the Stage (the clip moves left and right in a straight line):

```
this.onEnterFrame = function () {
  this._x = _level0._xmouse;
}
```

See Also

MovieClip._ymouse

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Chapter 18. ActionScript Language Reference

MovieClip._xscale Property — Flash 4

width of a clip or movie, as a percentage — ***read/write***

mc._xscale

Description

The floating-point _xscale property specifies the width of *mc*, relative to its *original width*, expressed as a percentage. If *mc* is an instance, its original width is the width of the instance's symbol in the Library. If *mc* is a main movie, its original width is the width of the movie at authoring time.

When the current width of *mc* is the same as its original width, _xscale is 100. An _xscale of 200 doubles *mc*'s width relative to its original width. An _xscale of 50 halves *mc*'s width relative to its original width.

The _xscale property scales a clip proportionately on the right and left sides of its registration point. To size a clip from one side only, place all of the clip's content on one side of the registration point in the clip's symbol (this is a useful technique for creating horizontal preloader bars). When a clip's _xscale is set to a negative value, the clip is flipped horizontally, as if across a vertical mirror running through its registration point (i.e., it becomes a mirror image of itself), after which the negative value is treated as a positive. To flip a clip horizontally without resizing it, set the clip's _xscale to -_xscale.

Example

```
// Double the width of ball_mc (the height remains unchanged)
ball_mc._xscale *= 2;

// Create a mirror image of ball_mc
ball_mc._xscale = -ball_mc._xscale;
```

See Also

MovieClip._yscale

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Chapter 18. ActionScript Language Reference

MovieClip._y Property — Flash 4

vertical location of a clip or movie, in pixels — *read/write*

mc._y

Description

The floating-point _y property always indicates the vertical position of *mc*'s *registration point*, but it is measured relative to one of three possible coordinate spaces:

- If *mc* resides on the main timeline, _y is measured relative to the *Stage's top edge*. For example, a _y of 20 indicates that the registration point of *mc* is 20 pixels below the Stage's top edge; -20 indicates 20 pixels above.
- If *mc* resides on another movie clip instance's timeline, _y is measured relative to the *registration point* of that parent instance. For example, a _y of 20 indicates that the registration point of *mc* is 20 pixels below its parent instance's registration point; -20 indicates 20 pixels above.
- If *mc* is the main movie, _y is the vertical offset of the entire *.swf* document, relative to the Stage's top edge. For example, a _y of 200 indicates that the contents of the Stage are offset 200 pixels below their author-time position; -200 indicates 200 pixels above.

The _y property (along with all vertical coordinates in Flash) increases downward and decreases upward?the opposite of the Cartesian coordinate system. Fractional _y values are approximated in Flash with antialiasing (blurring). Flash's basic unit of measure is a twentieth of a pixel (a "twip"), so the shortest distance a clip can be moved vertically is .05 pixels. Smaller increments are ignored:

```
trace(ball_mc._y);    // Displays: 0
ball_mc._y += .01;
trace(ball_mc._y);    // Still displays 0. The assignment had no effect.
```

If *mc* is contained by an instance that is scaled and/or rotated, the coordinate space it inhabits is also scaled and/or rotated. For example, if *mc*'s parent is scaled by 200% and rotated 90 degrees clockwise, _y will increase toward the left rather than downward, and a single unit of _y will actually be 2 pixels instead of 1.

Because switching between instance and main movie coordinate spaces can be cumbersome, the *MovieClip* object provides the *localToGlobal( )* and *globalToLocal( )* methods for performing coordinate-space transformations.

Example

The following *onEnterFrame( )* handler causes ball_mc to move down 5 pixels with each passing frame

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

(assuming that its coordinate space hasn't been altered by transformations to its parent):

```
ball_mc.onEnterFrame = function () {
  this._y += 5;
}
```

See Also

MovieClip.globalToLocal( ), *MovieClip.localToGlobal( )*, `MovieClip._x`

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Chapter 18. ActionScript Language Reference

MovieClip._ymouse Property — Flash 5

vertical location of the mouse pointer — ***read-only***

mc._ymouse

Description

The floating-point _ymouse property indicates the vertical location of the mouse pointer's hotspot, relative to the coordinate space of *mc*. If *mc* is a main movie, _ymouse is measured from the Stage's top edge. If *mc* is an instance, _ymouse is measured from the instance's registration point. To obtain a consistent _ymouse coordinate that is always measured relative to the Stage, use _root._ymouse.

Example

Placing the following code on a clip causes it to follow the mouse pointer's vertical position, relative to the Stage (the clip moves up and down in a straight line):

```
onEnterFrame = function () {
  this._y = _level0._ymouse;
}
```

See Also

MovieClip._xmouse

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Chapter 18. ActionScript Language Reference

MovieClip._yscale Property — Flash 4

height of a clip or movie, as a percentage — ***read/write***

mc._yscale

Description

The floating-point _yscale property specifies the height of *mc*, relative to its *original height*, expressed as a percentage. If *mc* is an instance, its original height is the height of the instance's symbol in the Library. If *mc* is a main movie, the original height is the height of the movie at authoring time.

When the current height of *mc* is the same as its original height, _yscale is 100. A _yscale of 200 doubles *mc*'s height relative to its original height. A _yscale of 50 halves *mc*'s height relative to its original height.

The _yscale property scales a clip proportionately above and below its registration point. To size a clip from above or below only, place all of the clip's content above or below the registration point in the clip's symbol (this is a useful technique for creating vertical preloader bars). When a clip's _yscale is set to a negative value, the clip is flipped vertically, as if across a horizontal mirror running through its registration point (i.e., it becomes a vertical mirror image of itself), after which the negative value is treated as a positive. To flip a clip vertically without resizing it, set the clip's _yscale to -_yscale.

Example

```
// Double the height of ball_mc (the width remains unchanged)
ball_mc._yscale *= 2;
// Flip ball_mc vertically
ball_mc._yscale = -ball_mc.yscale;
```

See Also

MovieClip._xscale

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Chapter 18. ActionScript Language Reference

NaN Global Property — Flash 5

constant representing invalid numeric data (Not-a-Number) — *read-only*

NaN

Description

NaN is a special numeric constant used to represent invalid numeric data (NaN is an acronym for "Not-a-Number"). Numeric operations that cannot be resolved to a legitimate number yield the value NaN. For example:

```
Math.sqrt(-1);      // An imaginary number that cannot be represented.
15 - "coffee cup";  // "coffee cup" cannot be converted to a number.
```

Note that NaN is still numeric data, even though it is not a calculable number:

```
typeof NaN;    // Yields "number"
```

Usage

The value NaN is hardly ever used directly in source code; rather, it serves as a way for an operation to return an error condition. Because NaN does not compare equal to itself, the only way you can detect it is with the global function *isNaN( )*. NaN is shorthand for Number.NaN.

See Also

isNaN( ), Number.NaN; Section 4.3.3

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Chapter 18. ActionScript Language Reference

new Operator — Flash 5

instantiate a new object

```
new constructor()
```

Operands

constructor

A class constructor function used to initialize the new object.

Returns

A reference to the newly created object.

Description

The *new* operator creates an instance of the class specified by `constructor`. For complete coverage of the role *new* plays in object-oriented programming, see Chapter 12.

See Also

`Object._ _proto_ _`; Section 5.12.2; Chapter 12

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Chapter 18. ActionScript Language Reference

newline Constant Flash 4

insert a line break

`newline`

Returns

A newline character.

Description

The constant `newline` represents a standard line break character (ASCII 10). It is synonymous with the escape sequence "`\n`" or the HTML tag `<BR>` and is used to force a line break in a block of text (usually for display in a text field variable).

Example

```
myOutput = "hello" + newline + "world";
```

See Also

`TextField``.htmlText`, *`TextField`*`.text`; Section 4.5.2.2

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Chapter 18. ActionScript Language Reference

nextFrame( ) Global Function — Flash 2

advance a movie or movie clip's playhead one frame and stop it

`nextFrame()`

Description

The *nextFrame( )* function moves the playhead of the *current timeline* ahead one frame and stops it there. The current timeline is the timeline from which the *nextFrame( )* function is invoked. The *nextFrame( )* function is synonymous with *gotoAndStop(_currentFrame + 1);*. If it is invoked on the last frame of a timeline, *nextFrame( )* simply stops the playhead on that frame, unless another scene follows the current scene, in which case *nextFrame( )* moves the playhead to frame 1 of the next scene.

See Also

gotoAndStop( ), *MovieClip.nextFrame( )*, *nextScene( )*, *prevFrame( )*

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Chapter 18. ActionScript Language Reference

nextScene( ) Global Function Flash 2

advance a movie's playhead to frame 1 of the next scene

```
nextScene( )
```

Description

The *nextScene( )* function moves the main playhead of a movie to frame 1 of the scene following the *current scene* and stops the main movie timeline. The current scene is the scene from which the *nextScene( )* function is invoked. It must be invoked on a scene's main timeline in order to have an effect; that is, *nextScene( )* does not work inside a movie clip instance. If it is invoked from the last scene in a movie, *nextScene( )* sends the playhead to frame 1 of that scene and halts movie playback.

> I recommend against using scenes when working with ActionScript-intensive movies. Unlike movie clips, scenes are not represented by objects and cannot be manipulated directly by most built-in functions. Normally it's better to use labels and movie clips as pseudo-scenes in your timeline instead of using Flash's scene feature.

See Also

nextFrame( ), prevScene( )

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Chapter 18. ActionScript Language Reference

Number( ) Global Function — Flash 5

convert a value to the Number datatype

```
Number(value)
```

Arguments

value

An expression containing the value to be converted to a number.

Returns

The result of converting `value` to a primitive number.

Description

The *Number( )* function converts its `value` argument to a primitive numeric value and returns that converted value. The results of converting various types of data to primitive numbers are described in Table 3-1 in Chapter 3. It's rarely necessary to use the *Number( )* function; ActionScript automatically converts values to the *number* type when appropriate.

Be sure not to confuse the global *Number( )* function with the *Number* class constructor. The former is a function that converts a value to the *number* type, whereas the latter is the class used to wrap a primitive numeric datum in an object that can take properties and methods.

Usage

Note that the *Number( )* function frequently appears in Flash 4 *.fla* files that have been converted to the Flash 5 format. For information on how datatypes are handled when Flash 4 files are converted to Flash 5, see Section C.3.4.

See Also

The *Number* class, *parseFloat( )*, *parseInt( )*; Section 3.4.2

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Chapter 18. ActionScript Language Reference

Number Class Flash 5

wrapper class for primitive numeric data

Constructor

`new Number(`*`value`*`)`

Arguments

value

An expression to be resolved and, if necessary, converted to a numeric value, then wrapped in a *Number* object.

Class Properties

The following properties are accessed directly as properties of the *Number* class, using `Number.`*`propertyName`* . To access them you do not need to instantiate a new *Number* object (i.e., there is no need for the constructor function). Some of these properties, such as `NaN` , don't even require the `Number.`*`propertyName`* notation. You can simply use `NaN` as shorthand for `Number.NaN` (details follow in entries for each property).

MAX_VALUE

The largest representable positive number in ActionScript.

MIN_VALUE

The smallest representable positive number in ActionScript.

NaN

Special Not-a-Number value indicating invalid numeric data.

NEGATIVE_INFINITY

Any number more negative than `-MAX_VALUE` .

POSITIVE_INFINITY

Any number larger than `MAX_VALUE` .

Methods

toString( )

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Convert a number to a string.

Description

The *Number* class has two purposes:

- It gives us access to built-in properties that represent special numeric values—`MAX_VALUE` , `MIN_VALUE` , `NaN` `NEGATIVE_INFINITY` , and `POSITIVE_INFINITY` —that can be used to check whether numeric data is va or within the acceptable range.
- It can be used to convert between different number systems, such as base-10 (decimal) and base-16 (hexadecimal). Refer to the *Number.toString( )* method for details.

Usage

There is no need to create a *Number* object if you simply want to access the numeric properties it defines. In fact, where applicable, it is easier to use global property equivalents (`NaN` , `Infinity` , and `-Infinity`). Frankly, it is rare that you'll need the *Number* class properties at all.

On the other hand, the *Number* class's *toString( )* method is used with a *Number* object. However, the interpreter takes car of creating a *Number* instance for us whenever we invoke a method on a primitive numeric value. For example, here we u *toString ( )* to convert a decimal number to its hexadecimal equivalent:

```
x = 102;
hex = x.toString(16);  // A Number instance is automatically and (temporaril
                       // created to "wrap" x for the sake of this operation
```

As this is a fairly rare task, you probably won't be using instances of the *Number* class much.

See Also

The *Math* object; "Section 4.1

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Chapter 18. ActionScript Language Reference

Number.MAX_VALUE Property — Flash 5

the largest representable positive number in ActionScript — ***read-only***

`Number.MAX_VALUE`

Description

The `MAX_VALUE` property returns the largest representable positive number in ActionScript (1.79769313486231e+308). It is convenient when you want to start with a very large value, as shown in the following Example. Any number greater than `MAX_VALUE` can't be represented by ActionScript and is therefore considered to be `POSITIVE_INFINITY` (i.e., `Infinity`).

Example

Here we are looking for a minimum value in an array. We initialize the `minVal` variable to `MAX_VALUE`, knowing that any subsequent value will be less than it:

```
var myArray = [-10, 12, 99];
var minVal = Number.MAX_VALUE;
for (var thisValue in myArray) {
  if (myArray[thisValue] < minVal) {
    minVal = myArray[thisValue];
  }
}
trace ("The minimum value is " + minVal);
```

See Also

`Infinity`, `Number.MIN_VALUE`, `Number.POSITIVE_INFINITY`; Section 4.3.3

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Chapter 18. ActionScript Language Reference

Number.MIN_VALUE Property — Flash 5

the smallest representable positive number in ActionScript — ***read-only***

```
Number.MIN_VALUE
```

Description

The `MIN_VALUE` property stores the smallest representable positive number in ActionScript, 4.94065645841247e-324 (not to be confused with `-Number.MAX_VALUE`, the most-negative number allowed in ActionScript).

See Also

`Number.MAX_VALUE`; Section 4.3.3

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Number.NaN Property

Flash 5

---

constant representing invalid numeric data (Not-a-Number) ***read-only***

```
Number.NaN
```

Description

The `Number.NaN` property is provided for ECMA-262 compatibility. See the entry for its more convenient global property equivalent, `NaN`.

See Also

isNaN( ), `NaN`

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Chapter 18. ActionScript Language Reference

Number.NEGATIVE_INFINITY Property — Flash 5

constant representing any number more negative than -Number.MAX_VALUE — *read-only*

```
Number.NEGATIVE_INFINITY
```

Description

The `Number.NEGATIVE_INFINITY` property is provided for ECMA-262 compatibility. You should instead use its more convenient global property equivalent, `-Infinity`.

See Also

`Infinity`, `-Infinity`, `Number.POSITIVE_INFINITY`

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Chapter 18. ActionScript Language Reference

Number.POSITIVE_INFINITY Property — Flash 5

constant representing any number greater than Number.MAX_VALUE — *read-only*

```
Number.POSITIVE_INFINITY
```

Description

The `POSITIVE_INFINITY` property is provided for ECMA-262 compatibility. You should instead use its more convenient global property equivalent, `Infinity`.

See Also

`Infinity`, `-Infinity`, `Number.NEGATIVE_INFINITY`

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Chapter 18. ActionScript Language Reference

Number.toString( ) Method — Flash 5

convert a number to a string

```
numberObject.toString(radix)
```

Arguments

radix

An integer between 2 and 36 specifying the base of the number system used to represent `numberObject` in st
This argument is optional; if omitted, it defaults to 10.

Returns

The value of `numberObject` converted to a string.

Description

The *toString ( )* method retrieves the value of a *Number* instance, converts that value to a string, and returns that string. W
optional `radix` argument to convert numeric values between different bases (e.g., binary, octal, decimal, and hexadecin
letters A-Z are used to represent digits with the values of 10-35, respectively, although ordinarily only A through F are use
represent the hex digits equivalent to 10 through 15).

To use *Number.toString ( )* with a primitive numeric value, surround the value with parentheses, as in:

```
(204).toString(16);
```

Example

```
x = new Number(255);
trace(x.toString());       // Displays: "255" (string version of decimal)
trace(x.toString(16));     // Displays: "ff" (string version of hexadecimal)
trace(x.toString(2));      // Displays: "11111111" (string version of binary

// Convert a hex literal to a decimal string
trace((0xFFFFFF).toString(10));       // Displays: "16777215" (a string)
// Convert a hex literal to a decimal integer
x = parseInt(0xFFFFFF).toString(10); // x equals 16777215 (a number), not "
```

See Also

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Number( ) , *Object.toString( )* , *parseInt( )* ; Section 4.3.1

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Chapter 18. ActionScript Language Reference

Object Class — Flash 5

the basis for all other classes and the basis for generic objects

Constructor

`new Object()`

Properties

constructor

A reference to the class constructor function used to create the object.

__proto__

A reference to the `prototype` property of the object's constructor function.

Class Methods

The following method is invoked through the *Object* class itself, not through an instance of the *Object* class:

registerClass( )

Assign a constructor for a movie clip symbol.

Methods

The following are instance-level methods of the Object class.

addProperty( )

Define a getter/setter property for an object or class.

hasOwnProperty( )

Check whether a property is defined directly on an object.

toString( )

Convert the value of the object to a string.

unWatch( )

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Remove an existing watchpoint.

valueOf( )

Retrieve the primitive value of the object, if one exists.

watch( )

Create a watchpoint to filter property assignment.

Description

The *Object* class is the base class of the ActionScript object model. *Object* is used for two general purposes: as a constructor for creating new, generic objects, and as a superclass upon which to base new classes. All classes in ActionScript, whether user-defined or built-in, are descendants of the *Object* class. All instances of all classes therefore inherit the properties and methods of *Object* (though some classes override those properties). However, the class method *Object.registerClass( )* is accessed directly as a method of the *Object* class's constructor, *Object*, not through an instance of the class.

To create a generic object of the *Object* class directly in our code without using a constructor, we can use an *object literal* just as we might use a string literal or an array literal. An object literal is a series of comma-separated property name/value pairs, enclosed in curly braces. Here's the general syntax:

```
{ property1: value1, property2: value2, property3: value3 }
```

The names of properties in an object literal must be legal identifiers, as described in Chapter 15. The values can be any valid expression. For example:

```
// An object literal with two numeric properties
myObject = { x: 30, y: 23 };
// Set the x property's value using a complex expression
myOtherObject = { x: Math.floor(Math.random() * 50 + 1) };
```

Because object literals always create generic, anonymous objects, they typically are used when we need object-formatted data temporarily, such as when invoking *Sound.setTransform( )*, *Color.setTransform( )*, or *MovieClip.localToGlobal( )*.

See Also

Chapter 12

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Chapter 18. ActionScript Language Reference

Object.addProperty Method Flash 6

define a getter/setter property for an object or class

```
someObject.addProperty(propertyName, getterFunction, setterFunction)
```

Arguments

propertyName

A string indicating the name of the property to add. The name must adhere to the rules for creating an identifier, ou under Section 15.5 in Chapter 15.

getterFunction

A "getter" function that returns `propertyName` 's current value. By convention, it should be named `get PropertyName` , where `PropertyName` matches the `propertyName` parameter. It can be either a cus function or a built-in method of the object, as shown under the following Example.

setterFunction

A "setter" function, expected to set the value of `propertyName` internally. By convention, it should be named `PropertyName` , where `PropertyName` matches the `propertyName` parameter. It can be either a cus function or a built-in method of the object, as shown under the following Example. Use `null` for `setterFunc` when the property is read-only.

Returns

A Boolean; `true` if the operation was successful, `false` if `propertyName` is not a valid identifier or if either `getterFunction` or `setterFunction` is not a function object.

Description

The *addProperty ( )* method defines a so-called "getter/setter" property, named `propertyName` , on `someObject` getter/setter property behaves ostensibly like a regular property, except that retrieval and assignment of its value is handle automatically by the two functions: `getterFunction` and `setterFunction` . This allows developers to provid convenient property-style access to complex operations. When `someObject .propertyName` is used in any expre (except on the left side of an assignment operation), the interpreter must determine and return the property's value. To do s interpreter executes the function specified by `getterFunction` , and its return value is used as `propertyName` For example:

```
// Create an object.
var gamePlayer = new Object();
// Add the getter/setter property. We make the property read-only by
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

```
// using null for the setFunction parameter.
gamePlayer.addProperty("score", getScore, null);
// Define the function to execute when gamePlayer.score's value is read.
// This isn't a realistic example, but see the next example for realistic us
function getScore () {
  return 1500;
}
// Read gamePlayer.score, which automatically calls the getScore() function
// and uses its return value (1500) as the value to print in the Output wind
trace(gamePlayer.score);
```

When *someObject*.*propertyName* is used on the left side of an assignment operation (on the left of the equals s interpreter must set the property's value. To do so, the interpreter executes the function specified by *setterFunction* it the new value (as specified in the property assignment statement) as an argument. Often, the argument is simply stored i for later retrieval by the *getterFunction*. For example:

```
// Create an object.
var gamePlayer = new Object();
// Add the getter/setter property, this time specifying setScore() as the
// function to execute when gamePlayer.score is written.
gamePlayer.addProperty("score", getScore, setScore);
// Here's our getterFunction (a.k.a., the "getter").
function getScore () {
  return gamePlayer.currentScore;
}
// Here's our setterFunction (a.k.a., the "setter"). The newly assigned
// property value is passed in as the only parameter, which we call newVal.
function setScore (newVal) {
  // Store the assigned property value internally in a separate property
  // for later retrieval by getScore().
  gamePlayer.currentScore = newVal;
}
gamePlayer.score = 2000;  // Here, setScore() is invoked with a newVal of 20
trace(gamePlayer.score);  // Here, getScore() is invoked. Displays: 2000
// Just to prove it's working behind the scenes, check the
// internal currentScore property value.
trace(gamePlayer.currentScore);  // Displays: 2000
```

The value passed to the *setterFunction* must not be written into a property with the same name *propertyName*.

The following code mistakenly assigns `newVal` to `score`, which collides with the getter/setter property of the same na

```
function setScore (newVal) {
  // OOPS! THIS BREAKS THE GETTER/SETTER PROPERTY!
  gamePlayer.score = newVal;
}
```

To remove a getter/setter property, simply delete it and the supporting getter/setter functions, as in:

```
delete someObject.propertyName
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

```
delete getterFunc;
delete setterFunc;
```

Getter/setter properties normally are used when a property value must be calculated internally by a method, or when the e property value differs from the value stored internally by the class. For example, suppose we want to define a *Ball* object's property. Externally, we want `color` to use the natural language string values "blue", "red", and "green". But internally our `color` property value to be stored as an RGB integer. We want the `color` property to be set like this:

```
redBall.color = "red";
```

But we want to store the equivalent of "red" internally as `0xFF0000` . And when the value of `redBall.color` is ret must internally convert the number `0xFF0000` back to the string "red". Here's how we create our natural language `col` property as a getter/setter:

```
// Create the Ball class constructor.
function Ball () {
  // Create the setter method for the color property.
  this.setColor = function (newVal) {
    // Convert the assigned value to lowercase.
    var theColor = newVal.toLowerCase();
    // Convert the natural name to an equivalent number, and store
    // it internally in the property colorNum.
    if (theColor =  = "red") {
      this.colorNum = 0xFF0000;
    } else if (theColor =  = "green") {
      this.colorNum = 0x00FF00;
    } else if (theColor =  = "blue") {
      this.colorNum = 0x0000FF;
    } else {
      // If the color specified isn't red, green, or blue, use black.
      this.colorNum = 0x000000;
    }
  }

  // Create the getter method for the color property.
  this.getColor = function () {
    // Convert the internal colorNum property to its equivalent natural
    // name, and return the name as the value of the color property.
    if (this.colorNum =  = 0xFF0000) {
      return "red";
    } else if (this.colorNum =  = 0x00FF00) {
      return "green";
    } else if (this.colorNum =  = 0x0000FF) {
      return "blue";
    } else {
      return "black";
    }
  }

  // Add the getter/setter color property.
  this.addProperty("color", this.getColor, this.setColor);
}
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

```
// Create a ball instance.
redBall = new Ball();

// Assign our ball a color. Note that the setter function accepts
// any mixture of uppercase and lowercase letters in the color name.
redBall.color = "RED";
// Check the color now. The getter function always returns lowercase names.
trace(redBall.color);  // Displays: red
```

Notice that the getter and setter functions are invoked as methods of the object containing the getter/setter property, so the to the object as `this` and access its properties and methods. So far, however, this code is inefficient because it assigns th *( )* and *getColor ( )* methods directly to every instance of *Ball* . Luckily, getter/setter properties can be added to a class cor `prototype` property, where a class's methods are normally defined. Here's the improved version of our `color` getter/s property, attaching method and property definitions to `prototype` :

```
// The Ball class
function Ball () {
  // Instance initialization code goes here...
}

// The setter function
Ball.prototype.setColor = function (newVal) {
  var theColor = newVal.toLowerCase();
  if (theColor =  = "red") {
    this.colorNum = 0xFF0000;
  } else if (theColor =  = "green") {
    this.colorNum = 0x00FF00;
  } else if (theColor =  = "blue") {
    this.colorNum = 0x0000FF;
  } else {
    this.colorNum = 0x000000;
  }
}

// The getter function
Ball.prototype.getColor = function () {
  if (this.colorNum =  = 0xFF0000) {
    return "red";
  } else if (this.colorNum =  = 0x00FF00) {
    return "green";
  } else if (this.colorNum =  = 0x0000FF) {
    return "blue";
  } else {
    return "black";
  }
}

// The getter/setter color property, defined on Ball.prototype
Ball.prototype.addProperty("color",
                           Ball.prototype.getColor,
                           Ball.prototype.setColor);
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

```
// Create a ball instance
redBall = new Ball();

// Assign and check our ball's color
redBall.color = "RED";
trace(redBall.color);  // Displays: red
```

Note that getter/setter properties are used entirely for convenience. It is perfectly legitimate to expose getter and setter met directly, without providing getter/setter property access to the values they manipulate. For example, our *Ball* class could si require the direct use of the *setColor ( )* and *getColor ( )* methods rather than providing the `color` getter/setter property. Conversely, it is legitimate in ActionScript, though considered poor object-oriented programming (OOP) practice, to acce internal `colorNum` directly. That is, one could retrieve and assign the `colorNum` property as `redBall.colorNum` or bypass getter and setter functions altogether.

Bugs

In Flash 6, getter/setter properties cannot be added directly to the `_global` object. They can, however, be added to obje defined on the `_global` object.

Example

The following nifty code—suggested by Guy Watson of http://www.flashguru.co.uk—creates a getter/setter `depth` prop movie clips. It provides an interface that's easier to use and remember than the built-in *getDepth( )* and *swapDepths( )* met

```
// Add the property.
MovieClip.prototype.addProperty("depth",
                                MovieClip.prototype.getDepth,
                                MovieClip.prototype.swapDepths);

// Try it out. Note that swapDepths( ) can accept either a number or a targe
this.createEmptyMovieClip("empty_mc", 15);
trace(empty_mc.depth);     // Displays: 15
empty_mc.depth = 20;       // Pass an integer to swapDepths(). Depth becomes
trace(empty_mc.depth);     // Displays: 20
empty_mc.depth = ball_mc;  // Trade depths with ball_mc (passed to swapDepth
```

See Also

MovieClip.getDepth( ) , *MovieClip.swapDepths( )* , `Object ._ _proto_ _` , *Object.watch ( )* ; Chapter 12

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Chapter 18. ActionScript Language Reference

Object.constructor Property — Flash 5

a reference to the class constructor function used to create the object — ***read/write***

someObject.constructor

Description

The `constructor` property stores a reference to the constructor function that was used to create *someObject*. For example, the `constructor` property of a *Date* object is the *Date* constructor function:

```
now = new Date();
now.constructor =  = Date;       // Yields true
```

Note that in JavaScript, the `constructor` property is always read from *someObject*'s constructor's `prototype` property (*someObject*.`_ _proto_ _.constructor`). In Flash Player 6 and Flash Player 5, the `constructor` property is written directly to *someObject* so that it always faithfully refers to *someObject*'s actual constructor function. For a complete discussion of this subtle deviation, see Section 12.7.3 in Chapter 12.

Usage

Overwriting an object's `constructor` property is not recommended, because it alters the natural structure of class inheritance.

See Also

Section 12.7.2 and Section 12.7.3

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Chapter 18. ActionScript Language Reference

Object.hasOwnProperty( ) Method Flash 6 (undocumented)

checks whether a property is defined directly on an object

```
someObject.hasOwnProperty(propName)
```

Arguments

propName

A string indicating the name of the property to check.

Returns

A Boolean; `true` if *`propName`* is a direct property or method of *`someObject`*, otherwise `false`.

Description

The undocumented *hasOwnProperty( )* method returns a Boolean indicating whether *`someObject`* directly defines a property or method with the name *`propName`*, in which case it is `true`; otherwise, it is `false`. It is used mostly during debugging to distinguish a property assigned directly to an object instance from a property inherited from a class via the prototype chain. For example:

```
// A Rectangle class
function Rectangle (w, h) {
  // Direct properties
  this.width = w;
  this.height = h;
}
// An inherited property
Rectangle.prototype.numSides = 4;

// A Rectangle instance
rect = new Rectangle(10, 100);
// Check direct property:
trace(rect.width);                          // Displays: 10
trace(rect.hasOwnProperty("width"));        // Displays: true

// Check inherited property:
trace(rect.numSides);                       // Displays: 4
trace(rect.hasOwnProperty("numSides"));     // Displays: false
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Notice that `numSides` is accessible through `rect`, but is not defined on `rect` itself, so *rect.hasOwnProperty("numSides")* returns `false`. However, it would also return `false` if the property wasn't defined anywhere (on either `rect` or its class).

Reader Exercise: Write a custom function, *hasProperty( )*, that returns the number 2 if *hasOwnProperty( )* is `true`, returns 1 if *hasOwnProperty( )* is `false` but the property is inherited from a class, and returns 0 if the property doesn't exist for the object. (Hint: Even a declared property can contain an `undefined` value, so use *hasOwnProperty( )* to check each object in the prototype chain. If you get to the *Object* class and *hasOwnProperty( )* still returns `false`, the property doesn't exist.)

Reader Exercise 2: Write a custom function, *showProperties( )*, that displays the names and values of all of an object's properties and indicates which are read through the prototype chain.

Reader Exercise 3: Write a custom function, *showMethods( )*, that is analogous to *showProperties( )* but displays the names of an object's methods instead of its properties. (Hint: use *instanceof* to determine if a property is an instance of the *Function* class).

The *hasOwnProperty( )* method can be used with getter/setter properties added via *Object.addProperty( )*. It is defined by the ECMA-262 specification but not officially documented by Macromedia.

See Also

instanceof, `Object._ _proto_ _`; Chapter 12

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Chapter 18. ActionScript Language Reference

Object._ _proto_ _ Property — Flash 5

a reference to the object's constructor's prototype property — *read/write*

someObject._ _proto_ _

Description

The _ _proto_ _ property stores a reference to the automatically-assigned prototype property of *someObject*'s constructor function, which is used to transfer properties down through a class hierarchy. An object's _ _proto_ _ property is mostly used internally by the interpreter to look up inherited properties for an object, but we can also use it to determine the class of an object, as shown in Example 12-5. Some developers also use _ _proto_ _ to establish inheritance between a subclass and its superclass, as discussed in Section 12.7.3 in Chapter 12.

> Note that the _ _proto_ _ property is formed using *two* underscore characters at its beginning and another *two* underscore characters at its end.

Usage

Overwriting an object's _ _proto_ _ property is not recommended, because it alters the natural structure of class inheritance.

See Also

Function.prototype, *instanceof*; Section 12.7.1

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Object.registerClass( ) Class Method Flash 6

assign a constructor for a movie clip symbol

```
Object.registerClass(symbolID, theClass)
```

Arguments

symbolID

A string specifying the linkage identifier of an exported movie clip symbol, as set in the Library under Options → Linkage.

theClass

The class constructor function to associate with `symbolID`. Must be a function object reference, not a string. Use `null` to unregister a class.

Returns

A Boolean; `true` if *registerClass( )* was invoked with the correct number of arguments, `false` if not. Note that Flash 6 returns `false` only when *registerClass( )* is invoked with the wrong number of arguments; it does not check whether the class was registered successfully.

Description

The *registerClass( )* method is part of the toolset used to create a movie clip subclass from a movie clip Library symbol. It connects an ActionScript class to an exported Library symbol. When instances of the symbol are placed on the timeline manually or programmatically via *attachMovie( )* or *duplicateMovieClip( )*, Flash automatically invokes `theClass` as the new instance's constructor function and sets the instance's class to `theClass`. When the constructor function runs, its `this` keyword points to the instance being created. For example, the following code creates a constructor function, *Vehicle( )*, and then links it to the movie clip symbol exported as `carSymbol` in the Library:

```
// The constructor function
function Vehicle () {
  // Set speed on the new clip
  this.pixelsPerFrame = this.pixelsPerSecond / 12;
}

// Link the constructor to carSymbol
Object.registerClass("carSymbol", Vehicle);
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

When instances of the `carSymbol` movie clip are created, they are constructed by the *Vehicle( )* function. Inside the function, the `this` keyword refers to the new instance. Notice, however, that *Vehicle( )* takes no parameters. Parameters are expected to be provided as object properties by *attachMovie( )* or *duplicateMovieClip( )* (using an *initObject*). For example, the `pixelsPerSecond` property is passed by *attachMovie( )* and is available when *Vehicle( )* executes. The *Vehicle( )* function uses `pixelsPerSecond` to calculate the speed of the instance in pixels per frame (assuming the movie's frame rate is 12 frames per second).

To ensure that instances of our *Vehicle* class inherit all the methods and properties of the *MovieClip* class, we must make *Vehicle* a subclass of *MovieClip*, as follows:

```
// Set Vehicle's superclass to MovieClip
Vehicle.prototype = new MovieClip();
```

Registered classes should always inherit from *MovieClip*. Once we've registered our class and set *MovieClip* as its superclass, we can create the rest of the class as usual. Here's the completed class:

```
// The constructor takes no parameters. They are optionally passed by
// attachMovie() and duplicateMovieClip() via the initObject parameter.
function Vehicle () {
  this.pixelsPerFrame = this.pixelsPerSecond / 12;
}

// Associate the Vehicle class with a Library symbol.
Object.registerClass("carSymbol", Vehicle);

// Set Vehicle's superclass to MovieClip.
Vehicle.prototype = new MovieClip();

// Add the goForward() method to the Vehicle class.
Vehicle.prototype.goForward = function () {
  this.onEnterFrame = function () {
    this._x += this.pixelsPerFrame;
  }
}

// Add the brake() method to the Vehicle class.
Vehicle.prototype.brake = function () {
  delete this.onEnterFrame;
}

// Make an instance. The Vehicle constructor is called automatically.
// The properties of our initObject (attachMovie()'s last argument)
// are set on the fastCar_mc instance before the constructor runs.
this.attachMovie("carSymbol", "fastCar_mc", 1, { pixelsPerSecond: 100 });

// Call a method.
this.fastCar_mc.goForward();
```

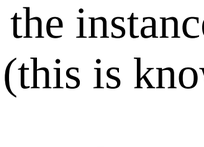
To remove a class registration, invoke *registerClass( )* using the ID of the movie clip to be unregistered as the `symbolID` parameter, and `null` as `theClass`. For example:

```
// Revert carSymbol to a normal movie clip. This sets carSymbol's
// class back to MovieClip instead of being a subclass of MovieClip.
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

```
// fastCar_mc remains an instance of Vehicle, however.
Object.registerClass("carSymbol", null);
```



Movie clips in existence when a symbol is unregistered remain instances of their registered class. The deregistering of a class affects subsequently created instances only.

Note that once a class is registered, instances of that class can immediately be created programmatically with *attachMovie( )* or *duplicateMovieClip( )*, as in our *Vehicle* example. However, instances placed manually on the timeline at authoring time must be created at least one frame after the class registration. Instances placed manually on the same frame as the class registration will not be members of the registered class. Use of the `#initclip` pragma can solve this problem.

For information on making a movie clip class self-contained (i.e., placing the class code on the timeline of the movie clip itself), see the `#initclip` pragma's entry in this *Language Reference*. For much more information on creating classes from movie clip symbols, see Chapter 14.

Note that a single class constructor can be registered to multiple Library symbols, but a single Library symbol cannot be registered to more than one class.

Though normally used with movie clips, *Object.registerClass( )* can also assign an arbitrary string identifier to a class constructor. Flash uses the identifier to reestablish the class of an object when it is imported into the Flash Player via *SharedObject*, *LocalConnection*, or Macromedia Flash Remoting MX (Flash Remoting). When Flash sends an instance of a registered class out of the Player, it includes the class identifier with the instance. When Flash later reads the instance back into the Player, it checks the instance's class identifier and reassociates the instance with its class (this is known as *casting* the object).

For example, suppose we create an instance, `p1`, of a custom class, *GamePlayer*, which defines a `score` property and an *addPoints( )* method, as follows:

```
// The class constructor
function GamePlayer () {
  // The score property
  this.score = 0;
}

// The addPoints() method
GamePlayer.prototype.addPoints = function (points) {
  this.score += points;
}

// The p1 instance
p1 = new GamePlayer();
p1.addPoints(100);
```

As the code is shown, `p1` is saved to disk via *SharedObject* (shown later) if its association with the *GamePlayer* class is lost, because *GamePlayer* has no identifier that links it to the serialized `p1` object. We can still retrieve the `p1` object and find its `score` property (because `score` is defined directly on `p1`), but we cannot access its class's *addPoints( )* method via the prototype chain. To maintain the relationship between `p1` and *GamePlayer*, we must add the following class registration, which assigns an arbitrary identifier ("GamePlayerClass") to the *GamePlayer* class:

```
Object.registerClass("GamePlayerClass", GamePlayer);
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Henceforth, all instances of *GamePlayer* will be marked transparently as belonging to the class with the identifier "GamePlayerClass" (this name is arbitrary, but by convention you should use the name of the class with the word "Class" appended). When Flash *serializes* `p1` before sending it out of the Player, it includes the identifier "GamePlayerClass" in the data to be sent. (*Serialization* is the process of converting an object's properties into a different format for transmission.) When Flash reads `p1` from an external source, it notices that a class identifier has been saved with the data. It then attempts to cast `p1` to the class registered to that identifier (*GamePlayer*). Technically, there is no guarantee that the *GamePlayer* class in effect when the *SharedObject* was saved is the same *GamePlayer* class in effect when the *SharedObject* is retrieved. Flash simply attempts to cast the retrieved object to a class whose registered identifier matches the object's registered identifier (which is assumed to be the same class constructor that was used when the data was first stored).

However, in the process of reconnecting the object with its class, Flash also executes the class constructor on the deserialized object. Normally, the object need not (indeed, should not) be reconstructed, so we must add to our constructor a flag that prevents any unwanted reinitialization. Here, then, is the basic code required to create a registered class:

```
// The GamePlayer class
function GamePlayer () {
  // Initialize the first time the constructor runs only
  if (this.inited) {
    return;
  }
  // Make a note that we've initialized this object
  this.inited = true;

  // Carry on with initialization
  this.score = 0;
}

// The addPoints() method
GamePlayer.prototype.addPoints = function (points) {
  this.score += points;
}

// Associate GamePlayer with the identifier GamePlayerClass
Object.registerClass("GamePlayerClass", GamePlayer);
```

And here is the code to create an object instance and save it as `player1` using a *SharedObject*:

```
// Create a GamePlayer instance
p1 = new GamePlayer();
p1.addPoints(100);
// Create the SharedObject
gameData_so = SharedObject.getLocal("gameData");
// Make a property to save our GamePlayer object
gameData_so.data.player1 = p1;
// Attempt to save the SharedObject
gameData_so.flush();
```

Here is the code required to retrieve a registered object from a *SharedObject*. There is no way to store an object's constructor or methods, even by registering its class. So, the *GamePlayer* class constructor and *addPoints( )* method must be defined in the receiving movie, and the class must be registered with *registerClass( )*. Note that, as

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

mentioned earlier, the *GamePlayer* constructor and method(s) need not be identical to the ones used at the time the object was saved, but ordinarily they would be. So, our receiving code begins identically to the code shown earlier for the sending movie. This code can be in the same movie as the sender (in which case it need not be repeated), or it can be in a different movie (perhaps even on the same machine, communicating with the sending movie via a *LocalConnection* object).

```
// The GamePlayer class
function GamePlayer () {
  // Initialize the first time the constructor runs only
  if (this.inited) {
    return;
  }
  // Make a note that we've initialized this object
  this.inited = true;

  // Carry on with initialization
  this.score = 0;
}

// The addPoints() method
GamePlayer.prototype.addPoints = function (points) {
  this.score += points;
}

// Associate GamePlayer with the identifier GamePlayerClass
Object.registerClass("GamePlayerClass", GamePlayer);
```

And here is the code to retrieve the *SharedObject* and recreate the object instance saved in `player1`. Note that the *GamePlayer* constructor is invoked when the *SharedObject.getLocal( )* function is executed (and the data is deserialized), although we don't access the `player1` instance object directly until it is assigned to the variable, `p1`.

```
// Retrieve the SharedObject.
gameData_so = SharedObject.getLocal("gameData");
// Retrieve our GamePlayer object.
p1 = gameData_so.data.player1;
// Check p1's score:
trace(p1.score);
// Add 50 points to p1's score. This works
// only because the GamePlayer class is registered.
p1.addPoints(50);
```

A similar workflow applies to objects sent over a network via *LocalConnection* and Flash Remoting. For more information on Flash Remoting, see:

http://www.macromedia.com/software/flash/flashremoting/

See Also

`#initclip`, *LocalConnection*, *MovieClip.attachMovie( )*, `Object._ _proto_ _`, the *SharedObject* object; Chapter 14

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Chapter 18. ActionScript Language Reference

Object.toString( ) Method Flash 5

the value of the object, as a string

```
someObject.toString()
```

Returns

An internally defined string that describes or otherwise represents the object.

Description

The *toString( )* method returns a string description for `someObject`. The ECMA-262 specification states that, by default, *Object.toString( )* should return the expression:

```
"[object " + class + "]"
```

where `class` is the internally defined name of the class to which `someObject` belongs. In practice, some ActionScript classes overwrite the default *toString( )* method of *Object* in order to provide more meaningful information about each instance of the class. For those classes that don't provide a custom *toString( )* method, ActionScript does not provide the individual class name. So, the default *toString( )* method for instances of most classes returns:

```
[object Object]
```

The ActionScript interpreter automatically invokes the *toString( )* method whenever `someObject` is used in a string context. For example:

```
x = new Object();
trace(x);           // Displays: "[object Object]"
```

We can implement a custom *toString( )* method when constructing our own classes. For example, the *Date.toString( )* method returns the date and time; the *Array.toString( )* method returns a comma-separated list of array elements; and the *Number.toString( )* method returns a string version of the number.

Example

This example shows how to provide a custom *toString( )* method for the *Ball* class that we created in Chapter 12:

```
// Make the Ball constructor
function Ball (radius, color, xPosition, yPosition) {
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

```
  this.radius = radius;
  this.color = color;
  this.xPosition = xPosition;
  this.yPosition = yPosition;
}

// Add a custom toString() method
Ball.prototype.toString = function () {
  return "A ball with the radius " + this.radius;
};

// Create a new Ball object
myBall = new Ball(6, 0x00FF00, 145, 200);

// Now check myBall's string value
trace(myBall);  // Displays: "A ball with the radius 6"
```

See Also

Array.toString( ), Date.toString( ), Number.toString( ), Object.valueOf( )

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Object.unwatch( ) Method Flash 6

remove an existing watchpoint

```
someObject.unwatch(prop)
```

Arguments

prop

A string indicating the name of the property that should no longer be watched. The name must adhere to the rules for creating an identifier outlined under Section 15.5 in Chapter 15.

Returns

A Boolean indicating whether `prop` was previously being watched; `true` if `prop` exists and was previously being watched; `false` otherwise.

Description

The *unwatch( )* method removes a watchpoint previously set on `prop` via the *Object.watch( )* method. Once the watchpoint is removed, value assignments to `prop` will no longer be moderated by the *watch( )* callback function. Note that a watchpoint can be removed only via *Object.unwatch( )*—deleting the watched property does not remove the watchpoint. For complete details, see *Object.watch( )*.

See Also

Object.watch( )

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Chapter 18. ActionScript Language Reference

Object.valueOf( ) Method Flash 5

the primitive value of the object

```
someObject.valueOf()
```

Returns

The internally defined primitive value of `someObject`, if a primitive value is associated with `someObject`. If no primitive value is associated with `someObject`, then `someObject` itself is returned.

Description

The *valueOf( )* method returns the primitive datum associated with an object, if such an association exists. This method is most commonly used with instances of the *Number*, *String*, and *Boolean* classes, which all have associated primitive data. It is rarely necessary to invoke *valueOf( )* explicitly; it is automatically invoked by the interpreter whenever `someObject` is used where a primitive is expected.

See Also

Boolean.valueOf( ), MovieClip.valueOf( ), Number.valueOf( ), Object.toString( ), String.valueOf( )

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Chapter 18. ActionScript Language Reference

Object.watch( ) Method Flash 6

create a watchpoint to moderate property assignment

```
someObject.watch(prop, callback, userData)
```

Arguments

prop

A string indicating the name of the property to watch on `someObject` . The name must adhere to the rules for creating an identifier, outlined under Section 15.5 in Chapter 15.

callback

A function called whenever a value assignment attempt is made on `prop` . The function return value becomes the value assigned to the property.

userData

An arbitrary data value passed to `callback` .

Returns

A Boolean; `true` if the watchpoint was created successfully; `false` otherwise (for example, if `callback` is not a va function).

Description

The *watch ( )* method registers a so-called *watchpoint* for `someObject.prop` . When a watchpoint is in effect, `cal` acts as a moderator for all value assignments to `someObject.prop` .

Whenever a value is assigned to `prop` , `callback` is executed and passed the following four arguments:

prop

The name of the property modified.

oldval

The old value of the property.

newval

The proposed new value of the property, as specified in the property assignment that triggered the watchpoint.

userData

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

An optional data value supplied by the original *watch( )* call.

Hence, `callback` must take the generalized form:

```
function someWatch (prop, oldval, newval, userData) {
  return theNewPropertyValue;
}
```

The `callback` function is expected to return the actual value to be assigned to the property. For example, the followin places on `rectangle.numSides` a watchpoint that rejects all `numSides` assignments by simply returning `oldva` (effectively making `numSides` read-only). This is known as *nullifying* the assignment.

```
// Create the object and numSides property
var rectangle = new Object();
rectangle.numSides = 4;
// Add the watchpoint
rectangle.watch("numSides", rectWatch);
// Create the callback function that moderates assignments to numSides
function rectWatch (prop, oldval, newval, userData) {
  // Return the old value, which is, therefore, reassigned to the property
  return oldval;
}
// Usage: Attempt to change the value of numSides
rectangle.numSides = 3;
// Check if we changed it...
trace(rectangle.numSides);  // Nope! Displays: 4
```

To allow the property assignment to proceed unaltered, `callback` can return `newval` . For example:

```
function rectWatch (prop, oldval, newval, userData) {
  // Return the proposed new value unscathed
  return newval;
}
```

A watchpoint callback function that returns `newval` , and therefore has no net effect on property assignment, can still be useful. By adding a *trace( )* command or a debugger breakpoint within the callback function, you can debug, log, or monitor any changes to the property.

Alternatively, `callback` can use `newval` to calculate or filter the property's final value (for example, by bringing ar of-range value into valid range). Or `callback` can perform other tasks related to the property assignment. For example consider a game in which a Buy button is disabled when the player has no money. By watching the player's `cash` propert can turn the Buy button on and off, depending on the players current amount of money:

```
// Simplified GamePlayer class
function GamePlayer (startingMoney) {
  // Add watchpoint (notice that watchpoints can be set before a property ex
  this.watch("cash", this.assignCash);
  // Assign the cash property an initial value
  this.cash = startingMoney;
}
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

```
// Method: assignCash()
GamePlayer.prototype.assignCash = function (prop, oldval, newval) {
  trace("Assigning cash: " + newval);
  // If the player has no cash...
  if (newval =  = 0) {
    // ...turn off the Buy button
    trace("Player has no money. Disabling Buy button.");
    gameGUI.buyButton.enabled = false;
  } else if (newval > 0) {
    // ...otherwise, turn on the Buy button
    trace("Player has some money. Enabling Buy button.");
    gameGUI.buyButton.enabled = true;
  }

  // Assign the new value to the cash property
  return newval;
}

// Make the player and issue 10 money units to start
p1 = new GamePlayer(10);

// Take away all the player's money
p1.cash = 0;

// OUTPUT:
Assigning cash: 10
Player has some money. Enabling Buy button.
Assigning cash: 0
Player has no money. Disabling Buy button.
```

A watchpoint can be set before a property exists, and the watchpoint continues to exist even if the watched property is del the property is recreated, the watch is triggered. To stop watching a property (i.e., to remove a watchpoint) use *Object.unv) .*

There are limitations on which properties can be watched:

- Properties inherited through an object's prototype cannot be watched. The watched property must be assigned direct `someObject` (use *Object.hasOwnProperty( )* to distinguish between direct and inherited properties).
- Properties available in Flash 4 (`_x` , `_y` , `_width` , `_height` , etc.) cannot be watched. See the Availability hea for each property's entry in this *Language Reference* .
- Getter/setter properties created with *Object.addProperty( )* cannot be watched. The getter/setter properties defined internally by Flash 6 ActionScript are listed in Table 18-15. Note that the *Camera* , *Microphone* , *NetStream* , and objects apply to Macromedia Flash Communication Server MX (Comm Server) and are not covered in this book (se http://www.macromedia.com/software/flashcom/ and Appendix A for additional resources).

Table 18-15. Getter/setter properties (Flash 6), which cannot be watched

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Object	Getter/setter properties
Button	tabIndex
Camera	All properties
Microphone	All properties
MovieClip	tabIndex
NetStream	All properties
Sound	duration position
Stage	All properties
TextField	All properties
TextFormat	All properties
Video	All properties
XML	contentType docTypeDecl ignoreWhite loaded status xmlDecl
XMLnode	attributes childNodes firstChild lastChild nextSibling nodeName nodeType nodeValue parentNode previousSibling

See Also

Object.addProperty ( ) , `Object ._ _proto_ _` , *Object.unwatch ( )*

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Chapter 18. ActionScript Language Reference

parseFloat( ) Global Function — Flash 5

extract a floating-point number from a string

`parseFloat(`*`stringExpression`*`)`

Arguments

stringExpression

The string from which a floating-point number is to be extracted.

Returns

The extracted floating-point number or, if extraction was unsuccessful, the special numeric value `NaN`.

Description

The *parseFloat( )* function converts a string to a floating-point number (a number with a fractional portion). It returns a useful value only for strings that contain a valid string representation of a floating-point number; otherwise, `NaN` is returned. The string must be of the following form:

- A valid starting character. Valid characters are +, -, 0 through 9, e, E, and ., although optional leading whitespace is acceptable (and ignored).
- Optional sign indicator (+ or -).
- At least one digit from 0-9 and an optional decimal point.
- Optional exponent starting with `e` or `E` followed by integer exponent, which can itself include an optional sign indicator (+ or -) — for example, "10e-3".

Trailing characters that cannot be parsed as part of the preceding numeric format are ignored but do not necessarily cause `NaN` to be returned. The number derived from the supplied string starts with the first nonblank character in the string and ends with the character before the first character that is not a legitimate character.

Usage

Because user-input data entered into text fields always belongs to the *string* datatype, we often use *parseFloat( )* to extract numeric data from user-supplied text. Note that *parseFloat( )* can extract numbers from strings that contain both numbers and nonnumeric characters, whereas *Number( )* cannot.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Examples

```
parseFloat("14.5 apples");    // Yields 14.5
parseFloat(".123");           // Yields 0.123
var x = "15b, 4, 23, 9";
parseFloat(x);                // Yields 15
```

See Also

isNaN( ), NaN, *Number( )*; Section 3.4.2; Section 4.2

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Chapter 18. ActionScript Language Reference

parseInt( ) Global Function Flash 5

extract an integer from a string, or convert numbers to base-10.

`parseInt(`*`stringExpression`*`)parseInt(`*`stringExpression, radix`*`)`

Arguments

stringExpression

The string from which an integer is to be extracted.

radix

An optional integer, between 2 and 36, specifying the base (or *radix*) of the integer contained in `stringExpre` not specified, the default radix depends on the contents of `stringExpression` (as described in the following

Returns

The extracted integer value, as a base-10 number regardless of the original `radix`; or, if extraction was unsuccessful, t numeric value `NaN`.

Description

The *parseInt ( )* function converts a string expression to an integer. It works only with strings that contain a valid string re an integer using the specified `radix`. The string must be of the following form:

- A valid starting character. Valid characters include +, -, and 0 through 9, although optional leading whitespace is ac ignored). Alphabetic characters (a-z and A-Z) are valid if the characters represent legitimate digits in the specified `r` example a-f and A-F are valid characters if `radix` is 16 (hexadecimal) but not if `radix` is 10 (decimal).
- Leading zero to indicate an octal number.
- Leading zero followed by an *X* or *x*, such as "0x" to indicate hexadecimal.
- Optional sign indicator (+ or -).
- One or more digits that are valid in the number system of the specified `radix`.

Trailing characters that cannot be parsed as part of the preceding numeric form are ignored but do not necessarily cause `N` returned.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

The number derived from the supplied string starts with the first nonblank character in the string and ends with the charac first character that is not a legitimate digit in the supplied `radix` . For example, in the base-10 system, the letter *F* is nc so the following expression yields the number 2:

```
parseInt("2F", 10);
```

But in the base-16 system (hexadecimal), the characters *A*, *B*, *C*, *D*, *E* , and *F* are valid numerals, so the following expres number 47:

```
parseInt("2F", 16);  // 2F in hexadecimal is 47 in base-10
```

The `radix` argument of *parseInt ( )* specifies the base of the number as it exists in the string. In other words, the `radi` says to the interpreter, "Treat this string as a base-16 (hexadecimal) number," or, "Treat this string as a base-2 (binary) nu

The *parseInt ( )* function also interprets the prefix `0x` to indicate a hexadecimal (a.k.a. *hex*) number, as if a `radix` of 1 specified, and a leading 0 (zero) to indicate an octal number, as if a `radix` of 8 was specified:

```
parseInt("0xFF");    // Parsed as hex, (implicit radix of 16) yields 255
parseInt("FF", 16);  // Parsed as hex, (explicit radix of 16) yields 255
parseInt("FF", 10);  // Parsed as decimal, (explicit radix of 10) yields NaN
parseInt("0377");    // Parsed as octal, (implicit radix of 8) yields 255
                     //  because (3 * 64) + (7 * 8) + (7 * 1) = 255
parseInt("377", 8);  // Parsed as octal (explicit radix of 8), yields 255
```

An explicit `radix` overrides any implicit `radix` :

```
parseInt ("0xFF", 10)   // Parsed as decimal, yields 0
parseInt ("0x15", 10)   // Parsed as decimal, yields 0 (not 15, and not 21)
parseInt ("0377", 10)   // Parsed as decimal, yields 377
```

Note that the *parseInt ( )* function extracts integer values only, unlike *parseFloat ( )* , which can also extract fractional valu used only with base-10 numbers.

Example

We use *parseInt ( )* primarily to extract integers from a string that contains both numbers and text or to remove the decima number (similar to *Math.floor ( )*):

```
parseInt("Wow, 20 people were there");    // Yields NaN
parseInt("20 people were there");         // Yields 20
parseInt("1001", 2);                      // Yields 9 (1001 evaluated in bin
parseInt(1.5);                            // Yields 1 (the number 1.5 is con
                                          // to the string "1.5" before the
                                          // operation proceeds)
```

See Also

Math.floor( ) , `NaN` , *Number( )* , *parseFloat( )* ; Section 3.4.2; Section 4.2

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Chapter 18. ActionScript Language Reference

play( ) Global Function — Flash 2

begin the sequential display of frames in a movie (i.e., play the movie)

```
play()
```

Description

Invoking the *play( )* function initiates the sequential display of the frames in the *current timeline*. The current timeline is the timeline of the main movie or movie clip from which the *play( )* function was invoked. Frames are displayed at a rate dictated by the frames per second (or FPS) setting of the entire movie, which is set under the Movie Properties (Modify Movie Frame Rate). If the movie is already playing, invoking *play( )* has no effect.

Once started, the playback of a movie or movie clip continues until another function invocation specifically stops the playback. All movie clips loop (begin playing again at frame 1) when the playhead reaches the end of the timeline. In a browser, however, main movies loop only if the code used to embed the movie in the HTML page specifies that the movie should loop, as determined by the `LOOP` attribute. If you're using the Publish command to embed your movie in an HTML page, set the `LOOP` attribute by selecting File Publish Settings HTML Playback Loop. See Appendix H for more details.

Example

Play the current timeline:

```
play()
```

Use the *MovieClip.play( )* function instead of the legacy Tell Target action to instruct a different movie clip to play.

Reader Exercise: Write a recursive function, named *allPlay( )*, that plays the current timeline and recursively plays all nested movie clips contained within the current timeline.

See Also

gotoAndPlay( ), *MovieClip.play( )*, *stop( )*

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Chapter 18. ActionScript Language Reference

prevFrame( ) Global Function Flash 2

send a movie's playhead back one frame and stop it there

```
prevFrame( )
```

Description

The *prevFrame( )* function moves the playhead of the *current timeline* back one frame and stops it there. The current timeline is the timeline of the main movie or movie clip from which the *prevFrame( )* function was invoked. The *prevFrame( )* function is synonymous with *gotoAndStop(_currentFrame - 1)*. If it is invoked on the first frame of a timeline, *prevFrame( )* simply stops the playhead on that frame, unless another scene precedes the current scene, in which case *prevFrame( )* moves the playhead to the last frame of the previous scene.

See Also

gotoAndStop( ), MovieClip.prevFrame( ), nextFrame( ), prevScene( ), stop( )

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Chapter 18. ActionScript Language Reference

prevScene( ) Global Function Flash 2

send a movie's playhead to frame 1 of the previous scene

```
prevScene( )
```

Description

The *prevScene( )* function moves the main playhead of a movie to frame 1 of the scene before the *current scene* and stops the main movie timeline. The current scene is the scene from which the *prevScene( )* function is invoked. It must be invoked on a scene's main timeline in order to have an effect; that is, *prevScene( )* has no effect inside a movie clip or *onClipEvent( )* handler. If invoked from the first scene in a movie, *prevScene( )* sends the playhead to frame 1 of that scene and halts movie playback there.

> I recommend against using scenes when working with ActionScript-intensive movies. Unlike movie clips, scenes are not represented by objects and cannot be manipulated directly by most built-in functions. Normally, it's better to use labels and movie clips as pseudo-scenes in your timeline instead of using Flash's scene feature.

See Also

nextScene( ), prevFrame( )

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Chapter 18. ActionScript Language Reference

print( ) Global Function Flash 5

print the frames of a movie or movie clip using vectors

```
print(target, boundingBox)
```

Arguments

target

A string or reference indicating the path to the movie clip or document level to print (references are converted to paths when used in a string context).

boundingBox

A string indicating how the printed frames of `target` should be cropped when printed. Cropping is defined by a bounding box that represents the entire printed page. The region of `target` included in the printed page can be s using one of three legal values for `boundingBox` (which must be a literal string):

"bframe"

The bounding box for each printed frame is set individually to match the size of each frame's contents. Hen every printed frame's content is scaled to fill the entire printed page. This maximizes the size of the printed images.

"bmax"

The area occupied by the content of all printed frames is combined to form a maximum bounding box. Each printed frame's content is scaled and placed on the printed page relative to the maximum bounding box. Tha is, each frame maintains its size relative to other frames, but all frames don't necessarily fill the printed page

"bmovie"

The bounding box for all printed frames is set to the size of a single, designated frame in the `target` clip Content of printed frames larger than the designated frame is cropped to the bounding box of the designated frame. Content of printed frames smaller than the designated frame is not scaled. To designate a frame as th one on which the bounding box should be based, assign it the label `#b` .

Description

Printing a Flash movie using a web browser's built-in File Print function behaves inconsistently and often results in poor quality printouts that don't use vector printing. In some browsers, the movie won't print at all, nor will most browsers print more than one frame at a time. Using the *print ( )* function, we can print the contents of one or more frames of a mov with accuracy and quality, directly from Flash. By default, *print ( )* causes all of the frames of `target` 's timeline to be printed, one frame per page, cropped or scaled according to the `boundingBox` argument. To designate specific frames

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

for printing (while excluding undesignated frames), we assign the label `#p` to the desired frames.

The *print ( )* function sends vectors directly to PostScript printers and sends vectors converted to bitmaps to non-PostScript printers. Because *print ( )* uses vectors, it cannot be used to print movies with alpha transparency or color transformations. To print movies that have color effects, use *printAsBitmap ( )* .

Usage

In Flash Player 4, the print functions are available via the Get Url Action (not available prior to build 4.0.20.0). For more details, see Macromedia's Flash Printing SDK, available at:

http://www.macromedia.com/software/flash/open/webprinting/authoring.html

Example

```
// Print every frame in the main movie timeline, scaling each to fill the pa
print("_root", "bframe");

// Print every frame in the main movie timeline,
// sizing each frame relative to the combined size of all frames
print("_root", "bmax");
```

When the `print_btn` button is clicked, Flash prints all frames in the button's parent timeline, cropped to the bounding box of the frame with the label `#b` , and sized to fill the page:

```
print_btn.onRelease = function () {
  print(this._parent, "bmovie");
}
```

See Also

getURL( ) , *printAsBitmap ( )* , *printAsBitmapNum ( )* , *printNum ( )*

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Chapter 18. ActionScript Language Reference

printAsBitmap( ) Global Function Flash 5

print the frames of a movie or movie clip as bitmaps

```
printAsBitmap(target, boundingBox)
```

Arguments

target

A string or reference indicating the path to the movie clip or document level to print (references are converted to paths when used in a string context).

boundingBox

A string—either `"bframe"`, `"bmax"`, or `"bmovie"`—indicating how the printed frames of `target` should be cropped or scaled when printed, as described earlier under *print( )*.

Description

The *printAsBitmap( )* function is functionally identical to *print( )*, except that it outputs rasterized content to the printer, not vectors. Therefore, it can successfully print movies with alpha channels and color transformations, but it produces poorer quality for vector-based artwork.

Usage

See Usage notes under the *print( )* function.

See Also

print( ), printAsBitmapNum( ), printNum( )

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

printAsBitmapNum( ) Global Function Flash 5

print the frames of a document level as bitmaps

```
printAsBitmapNum(level, boundingBox)
```

Arguments

level

A nonnegative integer or an expression that yields a nonnegative integer, indicating the document level to print.

boundingBox

A string—either `"bframe"`, `"bmax"`, or `"bmovie"`—indicating how the printed frames of `target` should be cropped or scaled when printed, as described earlier under *print( )*.

Description

The *printAsBitmapNum( )* function is nearly identical to *printAsBitmap( )*, except that it requires the target `level` of the print operation to be specified as a number rather than as a string. This means that *printAsBitmapNum( )* can print document levels only, not movie clips. It normally is used to dynamically assign the level of a movie to print, as in:

```
var x = 3;
printAsBitmapNum(x, "bmax");
```

which can also be achieved using string concatenation with the regular *printAsBitmap( )* function:

```
printAsBitmap("_level" + x, "bmax");
```

Usage

See Usage notes under the *print( )* function.

See Also

print( ), printAsBitmap( ), printNum( )

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

printNum( ) Global Function Flash 5

print the frames of a document level using vectors

```
printNum(level, boundingBox)
```

Arguments

level

A nonnegative integer or an expression that yields a nonnegative integer, indicating the document level to print.

boundingBox

A string—either `"bframe"`, `"bmax"`, or `"bmovie"`—indicating how the printed frames of `target` should be cropped or scaled when printed, as described earlier under *print( )*.

Description

The *printNum( )* function is nearly identical to *print( )*, except that it requires the target `level` of the print operation to be specified as a number rather than as a string. This means that *printNum( )* can print document levels only, not movie clips. It normally is used to dynamically assign the level of a movie to print, as in:

```
var x = 3;
printNum(x, "bmax");
```

which can also be achieved using string concatenation with the regular *print( )* function:

```
print("_level" + x, "bmax");
```

Usage

See Usage notes under the *print( )* function.

See Also

print( ), printAsBitmap( ), printAsBitmapNum( )

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Chapter 18. ActionScript Language Reference

_quality Global Property — Flash 5

the rendering quality of the Player — ***read/write***

`_quality`

Description

The `_quality` property stores a string that dictates the rendering quality of the Flash Player, as follows:

"LOW"
: Low quality. Neither bitmaps nor vectors are antialiased (smoothed).

"AUTOLOW"
: Start in low quality, but switch to high quality if the Player detects enough processing power.

"MEDIUM"
: Medium quality. Vectors are moderately antialiased using a 2 x 2 pixel grid.

"AUTOHIGH"
: Start in high quality, but drop to low quality if the Player detects poor performance (based on frame rate).

"HIGH"
: High quality (default). Vectors are highly antialiased using a 4 x 4 pixel grid. Bitmaps are antialiased when no animation is occurring.

"BEST"
: Best quality. Vectors are highly antialiased using a 4 x 4 pixel grid. Bitmaps are always antialiased.

Lower rendering quality means that fewer calculations are required to draw a frame of the movie, resulting in faster performance. The typical setting of `_quality` for most movies is "`HIGH`".

Example

Here we set the rendering quality of a movie to the best it can be (which also causes the slowest playback):

```
_quality = "BEST";
```

See Also

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

`_highquality`, *toggleHighQuality( )*

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Chapter 18. ActionScript Language Reference

random( ) Global Function — Flash 4; deprecated in Flash 5 in favor of Math.random( )

generate a random number

`random(`*`number`*`)`

Arguments

number

A positive integer.

Returns

A random integer between 0 and `number - 1` (inclusive).

Description

The deprecated *random( )* function was used in Flash 4 to generate random numbers. This function has been retired from the language and is available only for the sake of backward compatibility. In Flash 5 and later, use the preferred *Math.random( )* method. Note that *random( )* generated integers from 0 to `number - 1`, whereas *Math.random( )* generates floats from 0.0 to .999.

See Also

Math.random( )

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

removeMovieClip( ) Global Function Flash 4

delete a movie clip from the Flash Player

`removeMovieClip(`*`target`*`)`

Arguments

target

A string or reference indicating the path to the movie clip instance to be removed from the Player (references to movie clips are converted to paths when used in a string context).

Description

The *removeMovieClip ( )* function deletes the specified movie clip from the Player, leaving no trace of the clip's contents or shell. Subsequent references to the clip or any variables or properties it contained yield `undefined` .

The *removeMovieClip ( )* function can be used only on movie clip instances that were originally created via *duplicateMovieClip ( )* , *attachMovie ( )* , or *createEmptyMovieClip ( )* . It has no effect on movie clips on negative depths (normally those created at authoring time). To remove a negative-depth clip, first swap it to a positive depth, then remove it:

```
theClip_mc.swapDepths(1234);   // Pick a safe number to swap to.
theClip_mc.removeMovieClip();  // Remove the clip.
```

If more than one variable refers to the same movie clip, *removeMovieClip ( )* deletes the movie clip from the Stage but other references to it remain. That is, the movie clip persists as a data object until all references to the clip have been removed. For example:

```
// Create a new clip, clipInstance_mc, from the Library symbol clipSymbol.
// Store a convenient reference to clipInstance_mc in the variable c.
var c = this.attachMovie("clipSymbol", "clipInstance_mc", 1);
// At this point, two references to the movie clip exist:
// one in c, and one in clipInstance_mc.

// Remove clipInstance_mc
clipInstance_mc.removeMovieClip();
// Check if clipInstance_mc still exists...
trace(typeof clipInstance_mc);  // Nope. Displays: undefined
// Check if the reference to clipInstance_mc in c still exists....
trace(typeof c);                // Yup! Displays: movieclip
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

```
                                        // (the reference still exists!)
```

To fully remove all traces of the clip, we must delete any remaining variables that store a reference to it. Continuing the preceding example:

```
// Remove the reference in c.
delete c;
trace(typeof c);   // Displays: undefined (now it's really gone)
```

See Also

attachMovie ( ) , *duplicateMovieClip( )* , *MovieClip.removeMovieClip( )* , *MovieClip.swapDepths( )* , *unloadMovie ( )* , Section 13.6

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Chapter 18. ActionScript Language Reference

_root Global Property Flash 5 (same as "/" in Flash 4 movies)

a reference to the main timeline of the movie in the current level ***read-only***

```
_root
```

Description

The `_root` property stores a reference to the main timeline of the current document level. We use `_root` to invoke methods on the main movie or to retrieve and set the main movie's properties. For example:

```
_root.play();        // Play the main timeline
_root._alpha = 40;   // Make the whole movie partially transparent
```

The `_root` property can also be used as the starting point from which to access nested clips using dot syntax. For example:

```
_root.myClip.play();
_root.shapes.square._visible = false;
```

The `_root` property provides access to a movie clip in explicit terms. That is, a reference that starts with `_root` is valid (and invariant) from anywhere in a movie. However, note that `_root` is not necessarily synonymous with `_level`*n* (the main timeline of a given document level) or `_global` (the global variable space). For information on creating global variables, see `_global`.

> When used in a *.swf* file that is loaded into a movie cilp, the `_root` property changes meaning. For complete details, see "The _root and _parent properties" in Chapter 2.

See Also

`_level`*n*; `_parent`; "Accessing Variables on Different Timelines," and "The `_root` and `_parent` properties," in Chapter 2; "Referring to Main Movies with `_root` and `_leveln`," in Chapter 13

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Chapter 18. ActionScript Language Reference

scroll Property Flash 4

the current top line displayed in a text field ***read/write***

textFieldVar.scroll

Description

The `scroll` text field property can be both retrieved and set. This legacy property is accessed as a property of *`textFieldVar`*, an identifier associated with an on-screen text field. The *`textFieldVar`* identifier is specified in the Text Options panel (in Flash 5) or the Text Field Properties dialog box (in Flash 4). As of Flash 6, use the object-oriented *`TextField`*`.scroll` property instead, where *`TextField`* is an instance of the *TextField* class. See *`TextField`*`.variable` for more information.

A text field's `scroll` property indicates the number of the line currently displayed as the first line in the field's viewable region. When we set the value of `scroll`, it scrolls the text field, making the supplied line number the top line in the field's viewable region.

The `scroll` property normally is used with `maxscroll` to create text-scrolling interfaces, as described under *`TextField`*`.scroll`.

Bugs

When using text fields with embedded fonts in Flash Player 5.0.30.0, `scroll` may cause some text to be displayed outside the bounding box of the field. Some artifacts may not be removed as the text in the field scrolls. To work around the problem, use a mask over the text field layer. This problem is fixed in Flash Player 5.0.41.0 and later.

Example

```
// Sets x to the index of the top line displayed in myField
var x = myField.scroll;
// Scrolls the text in myField down one line
myField.scroll++;
```

See Also

maxscroll; *TextField*.scroll, *TextField*.variable

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Chapter 18. ActionScript Language Reference

Selection Object	Flash 5; enhanced in Flash 6 to support buttons and movie clips

control over text field selections and movie input focus

`Selection.`*`methodName`*`()`

Methods

addListener( )

Register an object to receive *onSetFocus( )* event notices.

getBeginIndex( )

Retrieve the index of the first selected character in the text field with keyboard focus.

getCaretIndex( )

Retrieve the index of the insertion point in a text field.

getEndIndex( )

Retrieve the index of the character following the selection in a text field.

getFocus( )

Identify the text field, button, or movie clip that is currently focused.

removeListener( )

Cancel event notices for the specified listener.

setFocus( )

Assign input focus to a text field, button, or movie clip object.

setSelection( )

Select characters in the currently focused text field.

Listener Events

onSetFocus( )

Occurs when input focus changes.

Description

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

We use the *Selection* object to control user interaction with text fields, to capture portions of text fields, and to detect input focus for buttons, movie clips, and text fields. *Selection* is a predefined object, not a class, so its methods are accessed directly, as in:

```
Selection.setFocus(input_txt);  // Focuses the input_txt text field
```

Character-manipulation methods of the *Selection* object always refer implicitly to the text field with keyboard focus, so we need only one *Selection* object. This is why, for example, *Selection.getBeginIndex( )* is accessed through the *Selection* object directly, not through an instance of the *TextField* class.

The methods of a *Selection* object can position the insertion point in the focused text field and select or retrieve a text field's content. These abilities add subtle but important behavior to user-input systems. For example, in a user-login screen, we can prompt a user to enter his name by placing the text insertion point in a name-entry text field. Or, we can highlight an error in a form by selecting the problematic text. Such control is also central to word processing-style applications.

Positions of the characters in a text field are referred to with zero-relative indexes, in which the first character is index 0, the second is index 1, and so on. Character indexing is described in detail under Section 4.6.4, in Chapter 4.

In Flash 6 the *TextField.replaceSel( )* method can be used to simulate cut, copy, and paste operations for a text field. Furthermore, the cut, copy, and paste shortcut keys (Ctrl-X, Ctrl-C, and Ctrl-V on Windows, or Cmd-X, Cmd-C, and Cmd-V on the Macintosh) are operational in Flash Player 6. The secondary mouse button (right-click on Windows, Ctrl-click on Macintosh) provides access to the cut, copy, and paste commands in Flash Player 5.

The *Selection* object can also control and report on keyboard focus (a.k.a. *input focus*) in a movie. When an object has keyboard focus, it accepts all keystrokes from the keyboard. For example, when a text field contains the text insertion point, it has keyboard focus, and keys pressed display as characters in the field. When a button has keyboard focus, it can be activated by pressing either the Enter key or spacebar. When a movie clip has keyboard focus, it can respond to keystrokes via its *onKeyDown( )*/*onKeyUp( )* event handlers. Buttons and movie clips are given keyboard focus either by the user via the Tab key (not by a mouseclick!) or programmatically via *Selection.setFocus( )*. By default, a focused button or movie clip is surrounded by a yellow rectangle (see the global `_focusrect` property). Changes in focus can be captured globally by the *Selection.onSetFocus( )* event or individually by the *onSetFocus( )*/*onKillFocus( )* event handlers for *Button*, *MovieClip*, and *TextField*. The currently focused item can always be determined with *Selection.getFocus( )*.

Usage

Clicking a form's Submit button automatically removes focus from any previously focused text field. To capture a text selection before focus is lost, use the rollover handler attached to the Submit button, because rollover events occur before the button is pressed. For example:

```
submit_btn.onRollOver = function () {
  // Note which field is focused before the button is pressed.
  focusedField = Selection.getFocus();

  // Note which characters are selected.
  beginIdx = focusedField.getBeginIndex();
  endIdx =   focusedField.getEndIndex();
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

```
}
```

By storing the focused field in a variable, we can then operate on the selected characters in that field or return focus to the field after, say, checking the user's input for errors.

See Also

The *Button* class, `_focusrect`, the *MovieClip* class, the *TextField* class

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Selection.addListener( ) Method Flash 6

register an object to receive onSetFocus( ) event notices

```
Selection.addListener(listener)
```

Arguments

listener

An object with an *onSetFocus( )* method defined.

Description

The *addListener( )* method adds `listener` to the list of objects notified when *Selection*'s *onSetFocus( )* event occurs. The `listener` argument is an object of any class that defines a method by the name *onSetFocus*; the method is invoked when the *onSetFocus( )* event occurs for *Selection*. To stop a `listener` object from receiving events, use *Selection.removeListener( )*.

Multiple objects can respond independently to each *onSetFocus( )* event. When multiple listeners are registered to *Selection*, their methods are invoked in the order the listeners were added.

For sample *Selection.addListener( )* code, see *Selection.onSetFocus( )*. For general information on how listeners and callback functions work, see Chapter 10.

See Also

Selection.onSetFocus( ), *Selection.removeListener( )*; Chapter 10

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Chapter 18. ActionScript Language Reference

Selection.getBeginIndex( ) Method Flash 5

retrieve the index of the first selected character in the text field with keyboard focus

```
Selection.getBeginIndex()
```

Returns

The index of the first character in the current selection (highlighted block of text). If no text field has keyboard focus, it returns -1. If a text field has focus but no characters are selected, it returns the value of *Selection.getCaretIndex ( )* .

Description

The *getBeginIndex ( )* method identifies the beginning of a selection. To determine the span of characters currently selected, use both *getBeginIndex ( )* and *getEndIndex ( )* .

Example

The following example creates a string representing the currently selected characters and then displays that string in the Output window:

```
var firstChar = Selection.getBeginIndex();
var lastChar  = Selection.getEndIndex();
// eval( ) is required because Selection.getFocus( ) returns a string
var currentSelection =
    eval(Selection.getFocus()).text.substring(firstChar, lastChar);
trace(currentSelection);
```

The following code extends the current selection by one character to the left:

```
Selection.setSelection(Selection.getBeginIndex()-1, Selection.getEndIndex())
```

See Also

Selection.getCaretIndex ( ) , *Selection.getEndIndex ( )* , *TextField.replaceSel ( )*

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Chapter 18. ActionScript Language Reference

Selection.getCaretIndex( ) Method Flash 5

retrieve the index of the insertion point in a text field

```
Selection.getCaretIndex()
```

Returns

The index of the insertion point in the currently focused text field. If no text field has keyboard focus, it returns -1. If the text field with focus is empty, it returns 0.

Description

The *getCaretIndex( )* method indicates the insertion point (i.e., the location of the cursor) in a text field. The cursor appears as an I-beam when a text field has keyboard focus. Use *setSelection( )* to set the location of the insertion point.

Example

Because *getCaretIndex( )* returns -1 when no text field has focus, we can determine whether any field has focus by checking whether *getCaretIndex( )* is equal to -1

The following example checks whether any text field has keyboard focus:

```
if (Selection.getCaretIndex() =  = -1) {
  trace("No field has focus");
}
```

Reader Exercise: Rewrite the preceding example as a Boolean function, *isTextFieldActive( )*, which returns `true` if a text field has focus and returns `false` otherwise. Modify it to accept an optional text field identifier and check whether that specific text field is active. (Hint: See *Selection.getFocus( )*).

See Also

Selection.getFocus( ), Selection.setSelection( )

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Chapter 18. ActionScript Language Reference

Selection.getEndIndex( ) Method Flash 5

retrieve the index of the character following the selection in a text field

```
Selection.getEndIndex()
```

Returns

The index of the character after the last character in the current selection (highlighted block of text). If no text field has focus, it returns -1. If a text field has focus but no characters are selected, it returns the value of *Selection.getCaretIndex ( )* .

Description

The *getEndIndex ( )* method identifies the end of a selection. To identify the span of characters currently selected, use bot *getEndIndex ( )* and *getBeginIndex ( )* .

Example

The following code extends the current selection by one character to the right:

```
Selection.setSelection(Selection.getBeginIndex(), Selection.getEndIndex()+1)
```

See Also

Selection.getBeginIndex ( ) , *Selection.getCaretIndex ( )*

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Selection.getFocus( ) Method

Flash 5; enhanced in Flash 6 to support buttons and movie clips

identify the currently focused button, movie clip, or text field

```
Selection.getFocus()
```

Returns

A string indicating the full path to the object that has keyboard focus (for example, `"_level1.theField_txt"` or `"_level0.listBox_mc")`. If no button, movie clip, or text field currently has keyboard focus, it returns `null`.

Description

The *getFocus( )* method identifies which button, movie clip, or text field (if any) currently has input focus by returning its string path. To turn that string into an object reference, we use *eval( )*. In the following example, we retrieve the instance name of the focused field using *getFocus( )*, and we convert that name into an object reference using *eval( )*. We then count the number of characters in the text field (not just the length of the selection) :

```
var numChars = eval(Selection.getFocus()).text.length;
```

Example

Because *getFocus( )* returns `null` when no object is selected, we can determine whether any object has focus by checking whether *getFocus( )* is equal to `null`, as follows:

```
if (Selection.getFocus() =  = null) {
  trace("No button, movie clip, or text field has focus");
}
```

See Also

eval( ), Selection.getCaretIndex( ), Selection.setFocus( )

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Chapter 18. ActionScript Language Reference

Selection.onSetFocus( ) Listener Event Flash 6

invoked when a movie's input focus changes

```
Selection.onSetFocus(oldFocus, newFocus)
```

Arguments

oldFocus

The *Button*, *MovieClip*, or *TextField* object that previously had focus, or `null` if no object was previously focused.

newFocus

The *Button*, *MovieClip*, or *TextField* object that now has focus, or `null` if no object is currently focused.

Description

The *onSetFocus( )* event is triggered whenever input focus in a movie changes—for example, when the user presses the Tab key to navigate from a form field to the form's Submit button. We can use *onSetFocus( )* to manage input focus from a central location. To handle focus for individual objects, we can alternatively use the *Button*, *MovieClip*, or *TextField* class's *onSetFocus( )* and *onKillFocus( )* handlers, which are triggered only when focus is given to or removed from a particular instance.

To respond to an *onSetFocus( )* event, we must first assign a callback function to the `onSetFocus` property of some object and then register that object as a listener. The *onSetFocus( )* callback should accept the `oldFocus` and `newFocus` parameters, as follows:

```
// Create a generic object
selListener = new Object();

// Assign a callback function to the object's onSetFocus property
selListener.onSetFocus = function (oldFocus, newFocus) {
  trace("Focus changed from " + oldFocus + " to " + newFocus);
}

// Register the listener with the Selection object
Selection.addListener(selListener);
```

Any number of listener objects of any class can respond to the *onSetFocus( )* event. To stop a listener from handling the *onSetFocus( )* event, use *removeListener( )*, as in:

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

```
Selection.removeListener(selListener);
```

Any *TextField*, *MovieClip*, or *Button* object can receive and lose focus when:

- The user navigates to or from it with the Tab key.
- The *Selection.setFocus( )* assigns focus to or from it programmatically.

In addition, a text field object can gain or lose focus when the user clicks in or out of it with the mouse. However, the `tabEnabled` property of *Button*, *MovieClip*, and *TextField* and the `selectable` property of *TextField* all affect input focus. For a detailed description of how and when buttons, movie clips, and text fields can be focused, see each class's *onSetFocus( )* entry.

Usage

The Tab key does not work reliably in Test Movie mode unless Control Disable Keyboard Shortcuts is enabled. Be sure to test keyboard behavior in the Standalone Player, if applicable, and in the Flash Player for all targeted web browsers.

See Also

`Button``.enabled`, *Button.onSetFocus( )*, *`Button`*`.tabEnabled`, *MovieClip.onSetFocus( )*, *MovieClip*`.selectable`, *`MovieClip`*`.tabEnabled`, *Selection.addListener( )*, *Selection.removeListener( )*, *Selection.setFocus( )*, *TextField.onSetFocus( )*, *`TextField`*`.tabEnabled`

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Chapter 18. ActionScript Language Reference

Selection.removeListener( ) Method Flash 6

cancels event notices for the specified listener

```
Selection.removeListener(listener)
```

Arguments

listener

An object that was previously registered as a listener for *Selection*.

Returns

A Boolean: `true` if `listener` was found and removed; `false` if `listener` was not registered as an event listener for *Selection*. The return value is useful for debugging code that removes listeners.

Description

The *removeListener( )* method stops `listener` from receiving *Selection* event notifications. It should be used only after `listener` has been registered as an event listener via *addListener( )*. The listener object is not deleted, but it can be removed and added repeatedly. Event listeners should be removed explicitly when the task they perform is no longer required.

See Also

Selection.addListener( ), *Selection.onSetFocus( )*; Chapter 10

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Selection.setFocus( ) Method

Flash 5; enhanced in Flash 6 to support buttons and movie clips

---

set input focus to a specific button, movie clip, or text field

```
Selection.setFocus(instanceName)
```

Arguments

instanceName

A string indicating the path to the button, movie clip, or text field instance to focus. Because instance references ar converted to paths when used in a string context, *instanceName* can also be an object reference, as in "`_level0.listBox_mc` " versus `_level0.listBox_mc` .

Returns

A Boolean indicating whether the focus attempt succeeded (`true`) or failed (`false`). A focus attempt fails if the *instanceName* is not found or if it is not a button, text field, or movie clip instance. The *setFocus( )* method also retu `false` if focus is disabled for *instanceName* , as explained in the following Description.

Description

The *setFocus ( )* method sets the input focus to the button, movie clip, or text field specified by *instanceName* .

If *instanceName* is a button, when focused, its Over frame is displayed, and it is surrounded by a yellow rectangle (u `_focusrect` is `false`). While focused, the button can be activated by pressing the Enter key or spacebar.

If *instanceName* is a movie clip, when focused, the frame labeled `_over` is displayed, if such a label exists, and th is surrounded by a yellow rectangle (unless `_focusrect` is `false`). While focused, the clip receives *onKeyDown ( )* *onKeyUp ( )* , *onMouseDown ( )* , and *onMouseUp ( )* event notices.

Take special note:

- In Flash 6, moving the mouse over the Stage removes button and movie clip focus, but not text field focus.
- Main movie timelines (e.g., `_root` and `_level1`) cannot receive keyboard focus.
- A movie clip cannot be focused via *Selection.setFocus( )* unless its `focusEnabled` property is `true` .
- A movie clip without button events cannot be focused via the keyboard unless its `tabEnabled` property is `true`

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

If *instanceName* is an input text field, the cursor appears in the field when it is focused, allowing characters to be en into the field. Dynamic and static text fields cannot be focused. However, a user-input text field with its `selectable` a `tabEnabled` properties set to `false` can still be focused with *setFocus ( )* .

Normally, an input text field is focused in order to draw attention to it or to allow the user to enter data. The *setFocus ( )* n can also provide a handy means of drawing a user's attention to erroneous input. In an online form, for example, a user ma mistype an email address. To alert the user to the error, we can set the focus to the email-address text field and ask her to f the problem. In Flash 5, we can also use *setFocus ( )* to create a custom Tab key order for the fields in a form, as shown in following Example. In Flash 6, tab order should be assigned directly via the *TextField* `.tabIndex` property.

Usage

Note that setting the focus of a text field automatically selects any content in that field. To set a field's focus without selec any characters, use the following code:

```
// First, focus theField
Selection.setFocus("theField_txt");

// Now place the insertion point at the start of theField
Selection.setSelection(0, 0);

// Or at the end of theField
Selection.setSelection(theField.length, theField.length);
```

> When a movie is viewed in a browser, a focused text field will accept text entry only if the Flash movie itself has focus (that is, if the user has clicked on the movie at some point during movie playback). Make sure a movie has focus before asking a user to type into one of its text fields. One way to do this is to include a button that says "Click Here to Start" at the beginning of your movie.

Example

This example shows how to assign a custom tab order to the fields in a fill-in form. While relevant only in Flash 5, the example shows the powerful control *Selection* provides over user input. The corresponding sample *.fla* file can be downlo from the online Code Depot. For more information on trapping the Tab key, see *TextField* `.tabIndex` .

```
// Custom tab order
// CODE ON THE MOVIE CLIP CONTAINING THE FORM FIELDS
onClipEvent (load) {
  // Store the path to this clip in a string. We'll use it
  // later when invoking Selection.setFocus( ) .
  path = targetPath(this);
  // Create an array with the names of our text fields, supplied in the
  // desired tab order. The first field listed is the default. In Flash 6,
  // these must be instance names, not Flash 5-style variable names.
  tabOrder = ["field1", "field3", "field2", "field4"];
}

onClipEvent (keyUp) {
  // If the Tab key was pressed...
  if (Key.getCode() =  = Key.TAB) {
    // ...If no field has focus...
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

```
if (Selection.getFocus() =  = null) {
  // ...then set focus to the first field in the array of fields.
  Selection.setFocus(path + "." + tabOrder[0]);
} else {
  // Otherwise, locate the currently focused field in the array of field
  i = 0;
  focused = Selection.getFocus();
  while (i < tabOrder.length) {
    // Extract the name of the field variable from the full path
    // that's returned by Selection.getFocus( ) .
    fieldName = focused.substring(focused.lastIndexOf(".") + 1);
    // Check each element of tabOrder  for the focused field.
    if (tabOrder[i] =  = fieldName) {
      // Stop when we find a match.
      currentFocus = i;
      break;
    }
    i++;
  }
  // Now that we know where the focused field lies in the tabOrder  arra
  // set the new focus to either the next field or the previous field,
  // depending on whether the Shift key is down.
  if (Key.isDown(Key.SHIFT)) {
    // Shift key is down, so go to the previous field, unless we're
    // already at the beginning, in which case go to the last field.
    nextFocus = currentFocus-1 =  = -1 ? tabOrder.length-1 : currentFocu
  } else {
    // Shift key is not down, so go to the next field, unless we're
    // already at the end, in which case go to the first field.
    nextFocus = currentFocus+1 =  = tabOrder.length ? 0 : currentFocus+1
  }
  // Finally, assign the new focus.
  Selection.setFocus(path + "." + tabOrder[nextFocus]);
 }
 }
}

// CODE ON BUTTON ON MAIN TIMELINE
on (keyPress "<Tab>") {
  // This placeholder code just traps the Tab key in Internet Explorer
  var tabCatcher = 0;
}
```

See Also

MovieClip .focusEnabled , *Selection.getFocus( )* , *Selection.setSelection ( )* , *TextField* .selectable
TextField .tabEnabled

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Selection.setSelection( ) Method Flash 5

select characters in the text field with focus, or set the insertion point

```
Selection.setSelection(beginIndex, endIndex)
```

Arguments

beginIndex

A nonnegative integer specifying the index of the first character to be included in the new selection.

endIndex

A nonnegative integer specifying the index of the character *after* the last character to be included in the new selection.

Description

The *setSelection( )* method selects (highlights) the characters from `beginIndex` to `endIndex - 1` in the text field with focus. If no field has focus, *setSelection( )* has no effect. It commonly is used to highlight problematic user input. To replace the text in the current selection, use *TextField.replaceSel( )*.

Usage

Though the *Selection* object does not have a *"setCaretIndex"* method, we can use the *setSelection( )* method to set the insertion point to a specific location within a text field. To do so, we specify the same `beginIndex` and `endIndex` values, as in:

```
// Set the insertion point after the third character
Selection.setSelection(3, 3);
```

Example

```
// Select the second and third letters of the currently focused text fielc
Selection.setSelection(1, 3);
```

See Also

Selection.getBeginIndex( ), Selection.getCaretIndex( ), Selection.getEndIndex( ), TextField.replaceSel( )

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

setInterval( ) Global Function Flash 6

execute a function or method every n milliseconds

```
setInterval(function, interval, arg1, arg2,...argn)
setInterval(object, method, interval, arg1, arg2,...argn)
```

Arguments

function

A reference to the function to execute.

interval

The amount of time between `function` or `method` executions, in milliseconds.

arg1 ,...arg n

A list of optional arguments to pass to `function` or `method` .

object

An object whose `method` will be executed every `interval` milliseconds.

method

The string name of the method to execute (must be defined on `object`).

Returns

An *interval identifier* , used to stop the timed execution of `function` or `method` (which is called "clearing the interval").

Description

The *setInterval ( )* function starts an *interval* , which executes the specified `function` or `method` every `interval` milliseconds. Here we start a very simple interval that executes the function *helloWorld ( )* once per second:

```
function helloWorld () {
  trace("Hello world!");
}
setInterval(helloWorld, 1000);  // 1000 milliseconds is 1 second
```

Notice that the function reference, `helloWorld` , is specified without the function call operator, ( ). The following attempt to set an interval fails because the call operator is included erroneously:

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

```
setInterval(helloWorld(), 1000);  // Oops! Use helloWorld, not helloWorld().
```

We can also pass a function literal to *setInterval ( )* . For example:

```
setInterval(function () {trace("Hello world!");}, 1000);
```

Arguments can be provided to the specified function after the `interval` parameter:

```
function hello (person) {
  trace("Hello, " + person);
}
setInterval(hello, 1000, "Mr. Thilgen");
```

However, the arguments provided are evaluated only once, when the interval is initialized. For example, the following code mistakenly attempts to position a movie clip under the mouse pointer by passing the pointer's location to the interval function:

```
function followMouse (clip, mousex, mousey) {
  clip._x = mousex;
  clip._y = mousey;
}
// Won't work because _root._xmouse and _root._ymouse are evaluated only onc
setInterval(followMouse, 100, trailer_mc, _root._xmouse, _root._ymouse);
```

In the corrected version, we retrieve data that changes between function invocations inside the function itself:

```
function followMouse (clip) {
  // Check _root._xmouse and _root._ymouse inside function
  clip._x = _root._xmouse;
  clip._y = _root._ymouse;
}
setInterval(followMouse, 100, trailer_mc);
```

The alternative syntax of the *setInterval ( )* function invokes a method on an object periodically. The following code executes the *nextFrame ( )* method of the main timeline (`_root`) every quarter-second, creating a slideshow effect:

```
setInterval(_root, "nextFrame", 250);
```

Remember to specify the method name as a string, not as a function reference!

When *setInterval ( )* is passed a function reference, the value of the `this` keyword within the function is `undefined` :

```
function showThis () {
  trace(this);
}
setInterval(showThis, 100);  // 'this' will be undefined
```

However, when *setInterval( )* is passed an object reference and a method name, the value of the `this` keyword is the object passed as the first argument:

```
setInterval(_root, "showThis", 100); // 'this' will be a reference to _root
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

When an interval is created, it is assigned an ID (interval identifier) that is returned by *setInterval ( )* . This ID should be stored in a variable so that the interval can later be stopped with *clearInterval( )* , as follows:

```
// Start the interval
var intervalID = setInterval(someFunction, 50);
// Stop the interval some time later
clearInterval(intervalID);
```

Often, the interval is cleared inside the function specified in the original *setInterval ( )* call:

```
var counter = 1;
function countToTen () {
  trace(counter);
  // If we've made it to ten...
  if (counter =  = 10) {
    // ...stop the interval
    clearInterval(counterInterval);
  }
  counter++;
}

// Start the interval
var counterInterval = setInterval(countToTen, 200);
```

If the interval ID is not stored, the interval can never be stopped. In Flash 6, interval IDs are integer values, but you should not stop intervals by guessing their numeric interval ID. The ID architecture is internal and subject to change in the future. To stop a group of intervals *en masse* , store their IDs in an array and use *clearInterval ( )* on each element.

Note that overwriting a variable that contains an interval ID does not clear the interval (which keeps running). Intervals can be cleared only via *clearInterval( )* . For example, in the following code, the variable `intervalID` is assigned two interval IDs, but both *functionA ( )* and *functionB ( )* continue to run every 100 ms, and `intervalID` stores the ID of the second interval only:

```
var intervalID = setInterval(functionA, 100);
var intervalID = setInterval(functionB, 100);
// The variable intervalID now contains the second interval ID, but
// the first interval still runs (and cannot be stopped!)
```

The *setInterval ( )* function provides a time-based alternative to the *onEnterFrame ( ) MovieClip* handler, which executes code once for every tick of the frame rate. However, *setInterval ( )* is still dependent on the frame rate of a movie. An interval can run 10 times per frame at most. So, even in a movie with a frame rate of 1 frame per second, an interval can still run only 10 times per second (every 100 ms) at most, even when a smaller `interval` value is specified. Theoretically, at 10 frames per second, an interval can run 100 times per second (10 times per frame x 10 frames per second), or every 10 ms. To update the screen within an interval function that runs more often that the frame rate, call *updateAfterEvent ( )* from within the function or method that the interval executes. For example:

```
function moveClip (clip) {
  // Move the clip
  clip._x += 5;
  // Update the screen
  updateAfterEvent();
}
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

```
setInterval(moveClip, 1, ball_mc);
```

When an interval is set to run less often than the duration of a tick of the frame rate, it will execute on the frame after the interval time has expired. For example, given an interval set to 400 ms and a frame rate of 10 fps (1 frame per 100 ms), Table 18-16 shows the frames on which the interval should execute.

Table 18-16. Interval execution relative to frame display

Frame number	Total time (milliseconds)	Interval executed?
0	0	no
1	100	no
2	200	no
3	300	no
4	400	yes
5	500	no
6	600	no
7	700	no
8	800	yes
9	900	no
10	1000	no

However, whether an interval runs more or less often than the frame rate of a movie, interval times are by no means guaranteed. In an actual test movie, the intervals in Table 18-16 may execute at frames 5 and 9, or even at frames 4 and 9. Code execution, Player rendering, and available system resources can all affect the actual timing of intervals. Where time is an important factor in an application (for example, in a game), functions should be written to account for variance in interval times.

Usage

There is no JavaScript-style *setTimeout ( )* function in ActionScript. To emulate *setTimeout( )* , create an interval function that clears itself the first time it runs, as shown in the following Example.

Example

The following code shows a simple timer that executes after 5 seconds and never executes again:

```
function timesUp () {
  trace("Time's up!");
  // Stop the timer
  clearInterval(timerInterval);
}
timerInterval = setInterval(timesUp, 5000);
```

See Also

clearInterval( ) , *MovieClip.onEnterFrame ( )*

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Chapter 18. ActionScript Language Reference

setProperty( ) Global Function Flash 4

assign a value to a movie clip property

```
setProperty(movieClip, property, value)
```

Arguments

movieClip

An expression that yields a string indicating the path to a movie clip. In Flash 5 and later, this may also be a movie clip reference because movie clip references are converted to paths when used in a string context.

property

The name of the built-in Flash 4 property to which `value` will be assigned. Must be an identifier, not a string (e.g., `_alpha`, not `"_alpha"`).

value

The new data value to be assigned to the specified `property` of `movieClip`.

Description

The legacy *setProperty( )* function assigns `value` to one of `movieClip`'s built-in Flash 4 properties. It cannot be used to set the value of custom (i.e., user-defined) properties or properties added to *MovieClip* in Flash 5 or later. In Flash 4, *setProperty( )* was the only means to assign movie clip property values; as of Flash 5, the dot and `[ ]` operators are the preferred means of setting both built-in and custom movie clip properties.

Example

```
// Flash 4 syntax: Rotate the main movie by 45 degrees:
setProperty("_root", _rotation, 45);
// Flash 5 and later syntax: Also rotates the main movie by 45 degrees:
_root._rotation = 45;
```

See Also

getProperty( ); "The `Objectness' of Movie Clips," in Chapter 13; Appendix C

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Chapter 18. ActionScript Language Reference

SharedObject Object Flash 6

local data storage and remote data transmission

Constructor

None; instances are created with *getLocal( )* or *getRemote( )*.

Properties

data

Container for the *SharedObject*'s data.

Methods

close( )

Closes the connection between a remote shared object and Comm Server. See the Comm Server documentation.

connect( )

Connects to a remote shared object on Comm Server. See the Comm Server documentation.

flush( )

Forces data to be written to disk.

getLocal( )

Returns a reference to a specified local *SharedObject* instance.

getRemote( )

Returns a reference to a remote *SharedObject* instance managed by Comm Server. See the Comm Server documentation.

getSize( )

Returns the size of the *SharedObject*, in bytes.

setFPS( )

Sets the frequency with which changes to a remote shared object are sent to Comm Server. See the Comm Server documentation.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Event Handlers

onStatus( )

Invoked when a *SharedObject* generates a status message (e.g., an error, information, or a warning).

Description

The *SharedObject* object provides tools for storing ActionScript data on the end user's computer and transferring data between Flash clients via Comm Server. When used for local data storage, the *SharedObject* is referred to as a *local SharedObject* (LSO). When used for remote data transfer, the *SharedObject* is referred to as a *remote SharedObject* (RSO). This section covers only the local data storage capabilities of *SharedObject*. Information on remote data transfer features can be found in the Comm Server documentation at:

http://www.macromedia.com/software/flashcom/

Like JavaScript cookies in a web browser, a local *SharedObject* saves information permanently on the local filesystem. When a movie is closed, the information is stored in a file. When the movie is reopened, the information is available once again. This allows movies to, say, record a player's high score and make it available across multiple game sessions. Local *SharedObject*s are more flexible than JavaScript cookies. They allow typed data to be stored and accessed directly through object properties rather than through an awkward cookie string of name/value pairs. Specifically, *SharedObject*s support the following datatypes: *number*, *string*, *boolean*, *undefined*, *null*, *array*, and *object* (custom classes plus *XML* and *Date*). Other built-in ActionScript classes and objects (such as *MovieClip* and *Function*) cannot be stored in a *SharedObject*.

We do not create *SharedObject* instances with a constructor function. Instead, to create a new local *SharedObject*, we use the *getLocal( )* method, which takes the name of the *SharedObject* to create as an parameter:

```
player1Info_so = SharedObject.getLocal("player1");
```

If a *SharedObject* by the specified name ("player1") already exists for the current movie, this code retrieves it from the local filesystem; if no such object exists, the Player creates it. Note the use of the suffix "`_so`"; this is Macromedia's official *SharedObject* suffix, which will enable code hinting in future versions of Flash.

After our shared object is created (or retrieved), we can add information to it by assigning custom properties to the built-in `data` property, such as:

```
player1Info_so.data.highScore = 1200;
```

When the movie is closed, the *flush( )* method is invoked, or the *SharedObject* is deleted and garbage-collected, Flash attempts to save the properties of `data` to a binary file with an extension of *.sol*. The save attempt fails if the size of the saved data exceeds the user-defined size limits, as discussed under *SharedObject.getLocal( )*. Therefore, you should always *flush( )* your data deliberately rather than relying on automatic saving, which does not allow you to handle size overages.

Once saved, the *SharedObject* can be retrieved later with the *getLocal( )* method, exactly as it was created:

```
// Load the SharedObject we created earlier
player1Info_so = SharedObject.getLocal("player1");
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Any properties defined on the object can be used immediately after *getLocal( )* is invoked to retrieve the LSO. For example, here we set the value of the `hiScore_txt` text field to the value of `player1Info_so.data.highScore`, which we created earlier:

```
hiScore_txt.text = player1Info_so.data.highScore;
```

By default, movies from any given domain can store a cumulative total of up to 100 KB of data on the user's computer. For information on requesting more space and handling storage requests that exceed the user's defined space limits, see *SharedObject.flush( )*.

For a prolonged example showing how to store address book contacts with a *SharedObject*, see the following article, written by me:

http://www.macromedia.com/desdev/mx/flash/articles/addressbook.html

Example

The following code creates a *SharedObject*, named `userProfile`, that tracks the total amount of time the user has spent viewing the movie:

```
var userProfile_so = SharedObject.getLocal("userProfile");
trace(userProfile_so.data.totalVisitTime);
var then = 0;

this.onEnterFrame = function () {
  var now = getTimer();
  var frameTime = now - then;
  userProfile_so.data.totalVisitTime += frameTime;
  this.then = now;
}
```

See Also

"Object Properties," in Chapter 12

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Chapter 18. ActionScript Language Reference

SharedObject.data Property — Flash 6

container for data stored locally or sent to Comm Server — ***read-only***

theSharedObj.data

Description

The data property contains a generic object whose properties are saved to disk when the movie is unloaded from the Player or when *SharedObject.flush( )* is invoked. If *theSharedObj* is a local *SharedObject* that has been retrieved from disk, the properties of data contain the values last saved. Properties can contain the following datatypes: *number*, *string*, *boolean*, *undefined*, *null*, *array*, and *object* (custom classes plus *XML* and *Date*). Other built-in ActionScript classes and objects (such as *MovieClip* and *Function*) cannot be stored in a *SharedObject*.

For a value to be saved, it must be stored within a property of data. Properties attached directly to *theSharedObj* are not saved to disk, while assignments to data itself are ignored completely. Therefore, you should attach properties to *theSharedObj*.data, not directly to *theSharedObj* itself, and don't try to overwrite the data property. For example:

```
timeTracker_so = SharedObject.getLocal("punchClock");

timeTracker_so.data.time = new Date();  // CORRECT! Will be saved.
timeTracker_so.time = new Date();       // INCORRECT! Will not be saved.
timeTracker_so.data = new Date();       // INCORRECT! Will be ignored.
```

Properties of data can be removed with the *delete* operator. For example:

```
delete theSharedObj_so.data.score;
```

Assigning a value of null or undefined to a property removes the value, but the property itself still exists and will still show up in a *for-in* loop and be saved to disk, albeit as an empty value.

The default maximum amount of data that can be saved to disk is 100 KB per domain. For details, see *SharedObject.flush( )*. To determine the size of the properties assigned to data, use *SharedObject.getSize( )*.

When a property of data is an object that belongs to a custom class, the save/retrieve process disconnects the object from its class. Methods and properties accessed via the prototype chain will not be available to the object after it is retrieved. However, we can use *Object.registerClass( )* to associate the object with an identifier used to restore its class at retrieval time. For details, see *Object.registerClass( )*.

Example

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

The following code retrieves a local *SharedObject* named "gameData". It then assigns a property, `player1score`, and saves the object by invoking *flush( )*:

```
// Create/retrieve the SharedObject.
gameData_so = SharedObject.getLocal("gameData");
// Make a property to save.
gameData_so.data.player1score = 2600;
// Attempt to save the SharedObject.
gameData_so.flush();
```

See Also

Object.registerClass( ), *SharedObject.flush( )*, *SharedObject.getLocal( )*, *SharedObject.getSize( )*

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

SharedObject.flush( ) Method Flash 6

forces data to be written (flushed) to disk

```
theSharedObj.flush()
theSharedObj.flush(minimumDiskSpace);
```

Arguments

minimumDiskSpace

An optional integer specifying the amount of local disk space requested for the current domain, in bytes. This allov to request more disk space than is required by `theSharedObj` . If `minimumDiskSpace` is omitted, *flush(* requests the amount of space returned by `theSharedObj` *.getSize( )* .

Returns

The string value "pending" if more disk space is required to store the object, or a Boolean: `true` if the save succeeded, `f` if it failed.

Description

The *flush ( )* method attempts to save `theSharedObj .data` 's properties to an *.sol* file, as described under `SharedObject` *.getLocal( )* . Functionally, it is similar to the automatic save that occurs when the movie is unloaded the Player, but it offers a movie the chance to ask the user for more disk space if required.

When *flush ( )* executes, Flash checks the local storage settings for the domain from which the movie was loaded. If local is set to "Never" for the domain, *flush ( )* simply returns `false` and the *SharedObject* is not saved. If the allowed local st space is greater than both the size of the *SharedObject* and the value of `minimumDiskSpace` , *flush ( )* returns `true` the *SharedObject* is saved. If the allowed local storage space is less than either the size of the *SharedObject* or the value o `minimumDiskSpace` , *flush ( )* returns "pending" and Flash prompts the user to increase the local storage space limit the user has either allowed or denied the request to increase the limit, `theSharedObj` *.onStatus( )* handler is invoked `infoObject` that indicates whether the save attempt succeeded.

The `minimumDiskSpace` argument is used to force Flash to request from the user more space than the *SharedObject* actually requires. This prevents Flash from repeatedly requesting space each time a *SharedObject* grows.

Example

The following code shows a typical Save button for a game. The game stores a large amount of game data in a *SharedObj* When the Save button is pressed, the movie reports save progress and, if necessary, asks the user to authorize space to sav

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

game. The *getSize ( )* method is used to show the user the game's space requirements. Notice that a complete save interfac requires an implementation for *onStatus ( )* :

```
// Create the SharedObject.
var gameData_so = SharedObject.getLocal("gameData");
// Assign a large map object as a property (assume mapObj holds the game dat
gameData_so.data.customUserMap = mapObj;
// Call saveGame() when save_btn is pressed.
save_btn.onRelease = saveGame;

// The saveGame() function attempts to save the shared object
function saveGame () {
  // Create a text field to tell the user what's happening.
  this._parent.createTextField("output_txt", 1, 0, 0, 300, 100);
  output_txt.border = true;
  output_txt.wordWrap = true;

  // Tell the user that saving has started.
  output_txt.text =  "Saving game...please wait...\n";
  // Request space based on getSize(). Divide by 1024 to convert bytes to KB
  // ask for 10 KB extra up front to prevent repeated requests for more room
  output_txt.text += "Space required: "
                  + (Math.floor(gameData_so.getSize() / 1024) + 10) + "KB.\n

  // Save the SharedObject and store the return value in a local variable.
  var saveAttempt = gameData_so.flush(gameData_so.getSize() + 10240);
  // If flush( ) returned...
  if (saveAttempt =  = true) {
    // ...true, the save was successful, and we're done.
    output_txt.text += "Game saved.\n";
  } else if (saveAttempt =  = "pending") {
    // ..."pending", Flash asks the user for more disk space.
    // The user's decision is reflected by the infoObject
    // passed to onStatus() when the user closes the settings dialog box.
    output_txt.text += "Please increase disk space to continue.\n";
  } else if (saveAttempt =  = false) {
    // ...false, the user has permanently disabled.
    // local storage for this domain. Ask the user to enable local storage.
    output_txt.text += "Save failed. To enable saving, "
                    +  "uncheck 'Never' under Local Settings.\n";
    // Just to make our point, open the local storage settings dialog box
    // where the user can reenable storage by unchecking the Never option.
    System.showSettings(1);
  }
}

// Callback invoked after a flush() overflow precipitated a
// "pending" condition. User has been prompted and either increased
// the domain's disk space limit or denied the request.
gameData_so.onStatus = function (infoObject) {
  // The infoObject.code property tells us what happened.
  if (infoObject.code =  = "SharedObject.Flush.Success") {
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

```
    // Yay! They gave us space!
    output_txt.text += "Required disk space added. Game saved.";
  } else if (infoObject.code =  = "SharedObject.Flush.Failed") {
    // Oh well. No space granted. We can't save the game.
    output_txt.text += "Insufficient disk space. Save failed.";
  }
}
```

See Also

SharedObject.getSize ( ) , SharedObject.onStatus ( ) , System.showSettings ( )

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Chapter 18. ActionScript Language Reference

SharedObject.getLocal( ) Method Flash 6

returns a reference to a specified SharedObject instance

```
theSharedObj.getLocal(objectName)
theSharedObj.getLocal(objectName, localPath);
```

Arguments

objectName

The string name of the local *SharedObject*. Can contain a forward slash (/). Cannot contain the following characters: ~ % & \ ; : " ' , < > ? #

localPath

An optional string specifying part of the local path to the *.sol* file saved on the client computer. Must match all or part of the actual path to the movie that creates `theSharedObj`.

Returns

A local *SharedObject* named `objectName`, or `null` if an illegal `localPath` was specified.

Description

The *getLocal( )* method is the means by which a local *SharedObject* is created and retrieved. When invoked, it searches for an .sol file named `objectName` at a location partially specified by `localPath`. The *.sol* file contains the data needed to recreate the ActionScript *SharedObject* object as it was last saved. If a matching *.sol* file is found, *getLocal( )* returns a local *SharedObject* with a `data` property populated with the values last saved to the *.sol* file. If no matching *.sol* file is found, *getLocal( )* creates and returns a new *SharedObject* named `objectName`. If the specified `localPath` is not valid, *getLocal( )* returns `null`, indicating that the operation failed.

When `localPath` is not specified, the *.sol* file is retrieved from an operating system-dependent location that mirrors the path to the movie that invoked *getLocal( )*. The path normally includes the movie name itself. For example, on my Windows XP system, a *SharedObject* named "player1", created by the movie *so.swf* at http://www.moock.org/asdg/so.swf, is retrieved from:

```
C:\Documents and Settings\fritz\Application Data\Macromedia\Flash
Player\moock.org\asdg\so.swf\player1.sol
```

Notice that *so.swf* actually is used as a directory name in the path to the *.sol* file. This prevents *.sol* filename conflicts amongst *SharedObject*s created by multiple movies in the same directory. For example, if the movies

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

asteroids.swf and *pong.swf* (posted at *moock.org/games/*) both use a *SharedObject* named "player1", then their locations on my computer are:

```
// Asteroids
C:\Documents and Settings\fritz\Application Data\Macromedia\Flash
Player\moock.org\games\asteroids.swf\player1.sol

// Pong
C:\Documents and Settings\fritz\Application Data\Macromedia\Flash
Player\moock.org\games\pong.swf\player1.sol
```

If both *asteroids.swf* and *pong.swf* want to share a single *SharedObject*, they must specify a common `localPath` when invoking *getLocal( )*. The `localPath` must contain part or all of the actual path to the *.swf* file, after the domain name. For example, *pong.swf* can use a `localPath` of "/", "/games", or "/games/pong.swf", resulting in the following file locations (again on my computer):

```
// For localPath "/", the location is:
C:\Documents and Settings\fritz\Application Data\Macromedia\Flash
Player\moock.org\player1.sol

// For localPath "/games", the location is:
C:\Documents and Settings\fritz\Application Data\Macromedia\Flash
Player\moock.org\games\player1.sol

// For localPath "/games/pong.swf", the location is:
C:\Documents and Settings\fritz\Application Data\Macromedia\Flash
Player\moock.org\games\pong.swf\player1.sol
```

Hence, any movie in the */games* directory on *moock.org* can access the *SharedObject* named "player1" by invoking *getLocal( )*, like this:

```
getLocal("player1", "/games");
```

And any movie on the entire *moock.org* site can access the *SharedObject* named "userInfo" by invoking *getLocal( )* like this:

```
// Use this notation to store globally available SharedObjects
getLocal("userInfo", "/");
```

However, a movie in the */products* directory cannot access a *SharedObject* created with a `localPath` of "/games".

> The local path of an .sol file is not always guaranteed to be human-readable (as it is in our examples). For a very long path, an .sol file might be named *.../domain/343345345345345.sol*. Hence, you should not rely on the local *.sol* file's location or name; rather, you should rely only on the *SharedObject* API to retrieve your data. The *getLocal( )* method's `localPath` parameter is unaffected by the internal name used by Flash to store the actual *.sol* file.

If a movie creates a *SharedObject* and is then moved to a new directory on the same domain, it will thereafter look for *SharedObjects* according to its new location, and it will not find any *SharedObjects* created at its old location unless a valid `localPath` is specified.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Movies running on the local filesystem (say, in the Standalone Player) are assigned the domain "localhost". For example, here is the location of *phoneNumbers.sol*, created by a movie running on the *root* of any hard drive on my computer:

```
C:\Documents and Settings\fritz\Application Data\Macromedia\Flash
Player\localhost\phoneNumbers.sol
```

Note that two local movies running from identical paths but on different local drives will conflict with each other because the *.sol* files are always stored on the system disk by default. For example, the following two movies can read and write each other's local *SharedObject*s:

```
C:\circle\demo.swf
D:\circle\demo.swf
```

To delete an .sol file from disk, delete all the properties assigned to the corresponding *SharedObject*'s `data` property. For example:

```
// Retrieve the SharedObject player1
player1Info_so = SharedObject.getLocal("player1");

// Delete all properties of player1Info_so.data
for (var p in player1Info_so.data) {
  delete playerInfo_so.data[p];
}
```

A small local storage limit setting in the Player does not prevent the creation of *SharedObject* instances in ActionScript. However, when the movie closes or when *flush( )* is invoked, the corresponding *.sol* file will not be created unless enough space is available. See *SharedObject.flush( )* and *System.showSettings( )*.

Bugs

In Flash 6, up through build 29 (version 6.0.29.0), when a movie is retrieved from a URL with a query string, the query string is included in the directory name that contains the local *.sol* file. For example, an *.sol* file saved from a movie loaded from:

http://www.moock.org/intro.swf?type=long

is stored erroneously in the local directory:

```
www.moock.org/intro.swf?type=long/
```

On Windows, question marks are not allowed in filenames, so the attempt to write the local *.sol* file fails in the preceding case. In later versions of the Flash Player (build 6.0.30.0 or later), the *.sol* file is stored correctly in:

```
www.moock.org/intro.swf/
```

When using a local *SharedObject* in a movie loaded with a query string, use the *localPath* parameter of *getLocal( )* to prevent the query string from being included in the directory name for the local *.sol* file.

Refer to the Flash Comm Server documentation for details on the *SharedObject* object's *getRemote( )*, *connect( )*, and *close( )* methods, which are used to manage remote shared objects.

See Also

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

SharedObject.getSize( ), System.showSettings( )

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Chapter 18. ActionScript Language Reference

SharedObject.getSize( ) Method Flash 6

returns the size of the SharedObject, in bytes

```
theSharedObj.getSize()
```

Returns

The integer size of `theSharedObj`, in bytes.

Description

The *getSize( )* method returns the total byte size of all the properties of `theSharedObj.data`. When requesting an increase in the local storage setting for a domain, the *getSize( )* method can be used to tell the user exactly how much space is required by a movie (see the Example for *SharedObject.flush( )*). To convert bytes to kilobytes (KB), divide by 1024:

```
soKB = Math.floor(theSharedObj.getSize() / 1024);
```

The more properties `theSharedObj.data` has, the longer *getSize( )* takes to execute.

See Also

SharedObject.flush( )

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Chapter 18. ActionScript Language Reference

SharedObject.onStatus( ) Event Handler Flash 6

invoked when a SharedObject generates a status message (error, information, warning)

theSharedObj.onStatus(*infoObject*)

Arguments

infoObject

An object whose `code` and `level` properties contain the status message.

Description

For local *SharedObject* s, the *onStatus ( )* event occurs only when *flush ( )* is invoked and the local storage limit for the domain is too low to save the *SharedObject* to disk. When the limit is too low, Flash displays a Settings dialog box asking user to increase the disk space for the current domain. Once the user's decision has been made, *onStatus ( )* is invoked and passed an *`infoObject`* that indicates whether the requested space was added. The *`infoObject`* defines two properties, `code` and `level` , with possible values pertaining to local *SharedObject* s listed in Table 18-17. Note that th is no "pending" status associated with *onStatus ( )* ; however, the word "pending" is a possible return value of the *flush ( )* method.

Table 18-17. SharedObject.onStatus( ) infoObject property values for local SharedObjects

code property values	level property values	Meaning
"SharedObject.Flush.Failed"	"error"	The user did not add enough space to store the *SharedObject* .
"SharedObject.Flush.Success"	"status"	The user added enough space to store the *SharedObject* .

For local *SharedObject* s, we need to check the value of the `code` property only, as shown in the following Example.

Example

The following *onStatus ( )* handler is executed after the user has decided whether to allow enough disk space to save `theSharedObj_so` . If enough space is not provided, the code disables the application:

```
theSharedObj_so.onStatus = function (infoObject) {
  // The infoObject's code property tells us whether the user
  // provided enough space to save the SharedObject.
  if (infoObject.code =  = "SharedObject.Flush.Success") {
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

```
    // Space was provided.
    output_txt.text = "Required disk space added. Application saved.";
  } else if (infoObject.code =  = "SharedObject.Flush.Failed") {
    // Space was not provided.
    output_txt.text += "You must increase disk space to use this application
    // Display an "application disabled" frame.
    gotoAndStop("disabled");
  }
}
```

See Also

SharedObject.flush ( ) , SharedObject.getSize ( )

About Safari | Terms of Service | Privacy Policy | Contact Us | Help | Submit a Problem

Copyright © 2002 O'Reilly & Associates, Inc. All rights reserved.
1005 Gravenstein Highway North
Sebastopol, CA 95472

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Chapter 18. ActionScript Language Reference

Sound Class Flash 6

control over sounds in a movie and external sound loading toolsFlash 5; enhanced in

Constructor

```
new Sound()
new Sound(target)
```

Arguments

target

A string indicating the path to the movie clip or document level whose sound is to be controlled. It can also be a reference to a movie clip or document level (references are converted to paths when used in a string context).

Properties

duration

The total length of the sound, in milliseconds.

id3

Access to the ID3 tags of an MP3 file.

position

The length of time the sound has played, in milliseconds.

Methods

attachSound( )

Associate a sound from the Library with a *Sound* instance.

getBytesLoaded( )

The number of downloaded bytes of an external sound file.

getBytesTotal( )

The physical disk size of an external sound file, in bytes.

getPan( )

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Retrieve the current pan setting.

getTransform( )

Determine the current distribution of the channels of a sound to the left and right speakers (i.e., balance).

getVolume( )

Retrieve the current volume.

loadSound( )

Download an external sound file into the Player.

setPan( )

Set the pan across a sound's left and right channels.

setTransform( )

Distribute the left and right channels between the left and right speakers (i.e., balance).

setVolume( )

Set the sound volume.

start( )

Start playing an attached or loaded event sound.

stop( )

Silence all sounds or silence a specified attached or loaded sound.

Event Handlers

onLoad( )

Occurs when an external sound finishes loading.

onSoundComplete( )

Occurs when a sound finishes playing.

Description

Objects of the *Sound* class are used to load external sounds and control both existing and loaded sounds in a movie. *Sound* objects can control:

- All the sounds in the Flash Player
- All the sounds in a particular movie clip instance or main movie (including all sounds in any nested clips)
- An individual programmatically attached or downloaded sound

To create a *Sound* object that controls all the sounds in the Player (including sounds in *.swf* files on document levels), use the *Sound* constructor without any parameters. For example:

```
myGlobalSound = new Sound();
```

To create a *Sound* object that controls all the sounds in a particular clip or main movie, supply a `target` parameter

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

indicating the clip or movie to control. Note that this also controls sounds in clips nested within `target` . For example:

```
spaceshipSound = new Sound("spaceship");  // Control sounds in spaceship cli
mainSound      = new Sound("_root");      // Control sounds on main timeline
```

To make an individual sound that can be started, stopped, and looped independently, create any kind of *Sound* object, and then add a sound to it using the *attachSound ( )* or *loadSound ( )* method.

To load an external sound file into a *Sound* object, use *loadSound ( )* . A loading sound's progress can be monitored with *getBytesLoaded ( )* , *getBytesTotal ( )* , and *onLoad ( )* .

Flash does not provide access to, or control over, sound frequency or pitch information, but Flash does expose the length of a sound (`duration`) and the elapsed time (`position`).

Flash can play a maximum of eight individual sounds simultaneously. When loaded via *attachSound( )* , each simultaneou sound needs its own *Sound* instance.

A sound does not stop playing when the object that controls it is deleted. Use *Sound.stop( )* to halt the playback of loaded or attached sounds.

See Also

`capabilities.hasAudio` , *stopAllSounds ( )* ; `_soundbuftime`

About Safari | Terms of Service | Privacy Policy | Contact Us | Help | Submit a Problem

Copyright © 2002 O'Reilly & Associates, Inc. All rights reserved.
1005 Gravenstein Highway North
Sebastopol, CA 95472

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Chapter 18. ActionScript Language Reference

Sound.attachSound( ) Method Flash 5

associate a sound from the Library with a Sound object

```
soundObject.attachSound(linkageIdentifier)
```

Arguments

linkageIdentifier

The name of the sound to attach, as specified in the Library under Options Linkage.

Description

The *attachSound( )* method places an exported Library sound under `soundObject`'s control. Once attached, the sound can be started and stopped individually by invoking *start( )* and *stop( )* on `soundObject`. For instructions on pausing a sound, see the online technote posted at:

http://www.moock.org/asdg/technotes/pauseSound/

In order for a sound to be attached to `soundObject`, the sound must be exported from the movie's Library. To export a sound, follow these steps:

1. In the Library, select the sound to export.
2. Select Options Linkage. The Linkage Properties dialog box appears.
3. Under Linkage, select Export For ActionScript.
4. In Flash 6, optionally uncheck the "Export in first frame" checkbox.
5. In the Identifier box, enter a unique name by which to identify the sound.

> In Flash 5, *all* exported sounds are loaded in the first frame of the movie that contains them (not when they are actually attached or played via ActionScript), which can cause long load delays if the sounds are large.

In Flash 5, you can gain better control over the loading of sounds by placing them in external *.swf* files and using *loadMovie( )* or *MovieClip.loadMovie( )* to import them as necessary.

In Flash 6, attached sounds can be exported on any frame. In the Linkage Properties dialog box, uncheck "Export in first frame," and then place the sound on the movie's main timeline at the desired export frame. The *attachSound( )* method will work only after that frame. Alternatively, use *loadSound( )* to retrieve an external sound file when one is needed.

Only one sound can be attached to a *Sound* object at a time, but you can create up to eight simultaneous *Sound*

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

objects. Attaching a new sound to a *Sound* object replaces any sound previously attached to that object. To keep multiple attached sounds in synch, attach them to separate *Sound* objects and start them all in a single frame, then use *Sound.setVolume( )* to silence each sound or make it audible as necessary.

When a movie with an exported sound is loaded into a target level or movie clip, the exported sound can be attached only if the *Sound* object's specified *target* matches that level or clip. For example, suppose we've stored exported sounds in *linkedSounds.swf*, and then we load *linkedSounds.swf* into a movie clip named host. When the *.swf* file has completely loaded, we can safely attach and play its exported sounds as follows:

```
// Create a new sound with a target of host
host.bang = new Sound(host);
host.bang.attachSound("loudBang");
host.bang.start();
```

It's a common mistake to omit the *target* argument when constructing *Sound* objects in loaded movies. If *target* is not supplied, it defaults to _level0, where the exported sound cannot be found. For example, the following attempt to play a sound in host fails because the sound loudBang is not available in the Library of _level0:

```
// Create a new sound with an implied target of _level0.
host.bang = new Sound();

// This operation fails..."loudBang" is not in _level0's Library.
host.bang.attachSound("loudBang");

// The sound won't start because it wasn't successfully attached.
// We forgot to specify a target when we constructed host.bang.
host.bang.start();
```

From within a movie that will be loaded into other movies, we can ensure that exported sounds will always be attachable by using the this keyword as the *target* parameter of all *Sound* objects. For example:

```
// Code inside sounds.swf, which will be loaded into other movies
music = new Sound(this);  // Set target to current movie clip
music.attachSound("song");
music.start();
```

Example

The following example adds a sound with the identifier phaser to the *Sound* object *phaserSound*. It then starts and stops the phaser sound. We need not specify the *target* in the constructor, provided this is the main move and not a movie loaded into a movie clip:

```
phaserSound = new Sound();
phaserSound.attachSound("phaser");

// Start the phaser sound
phaserSound.start();

// Stop just the phaser sound
phaserSound.stop("phaser");
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

See Also

Sound.loadSound( ), Sound.start( ), Sound.stop( )

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Chapter 18. ActionScript Language Reference

Sound.duration Property — Flash 6

the total length of the sound, in milliseconds ***read-only***

soundObject.duration

Description

The `duration` property returns the length, in milliseconds, of the attached or loaded sound associated with *`soundObject`* . It does not apply to timeline sounds placed at authoring time. If multiple sounds are loaded into a single *`soundObject`* , the `duration` property reflects the most recently loaded sound. If the sound is looped, `duration` returns the length of one loop. If no attached or loaded sound exists, `duration` returns `undefined` . While an external streaming sound is loading, `duration` reflects the length of the sound loaded so far. While an external event sound is loading, `duration` stores 0 until the sound has completely loaded, at which point it stores the actual length of the sound. Theoretically, `duration` should be available from *Sound.onLoad( )* , but a bug in Flash 6 prevents access to it (see the following Bugs).

The `duration` property can be used with `position` to determine the percentage of a sound that has played so far. See *`Sound .position`* for example code that increases the volume of a sound proportionate to the percentage played.

Bugs

In Flash 6, when a sound is exported from the Library and *loadSound ( )* is invoked on *`soundObject`* , the `duration` property matches the Library sound's length until part of the external sound has been downloaded to the Player. To ensure an accurate reading for `duration` , test its value only after *getBytesLoaded ( )* returns a positive value.

In Flash 6, the `duration` property is not initially available within *onLoad ( )* . Use *setInterval ( )* to poll for a valid `duration` , as follows:

```
song = new Sound();
song.onLoad = function (success) {
  // Because of the bug, no duration displays
  trace("Initial Duration: " + this.duration);

  // Store a reference to the current sound
  var theSound = this;

  // Create a function that checks if duration exists
  function pollForDuration () {
    if (theSound.duration >= 0) {
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

```
        // Show the duration, now that it's available
        trace("Actual duration: " + theSound.duration);
        // Play the sound
        theSound.start();
        // Stop polling for a valid duration
        clearInterval(intID);
      }
    }

    // If the sound loaded properly...
    if (success =  = true) {
      // Repeatedly check for a valid duration
      var intID = setInterval(pollForDuration, 100);
    }
}

// Load the MP3 file
song.loadSound("track1.mp3", false);
```

Example

The following code attaches the exported sound "JingleBells" to a *Sound* object and then checks the length of the sound:

```
music = new Sound();
music.attachSound("JingleBells");
trace(music.duration);  // Displays length of JingleBells, in milliseconds
```

See Also

Sound .position

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Chapter 18. ActionScript Language Reference

Sound.getBytesLoaded( ) Method Flash 6

the number of downloaded bytes of an external sound file

```
soundObject.getBytesLoaded()
```

Returns

An integer representing the number of bytes that have finished downloading to the Player. Divide by 1024 to convert byte to kilobytes (KB).

Description

The *getBytesLoaded ( )* method tells us the number of bytes of an external sound file that have downloaded into the Flash Player due to a *loadSound ( )* call. For a new *Sound* object, *getBytesLoaded ( )* returns `undefined` . Immediately after call to *loadSound ( )* , *getBytesLoaded ( )* also returns `undefined` . As data arrives, *getBytesLoaded ( )* returns the nur of bytes transferred so far (from 0 to *getBytesTotal ( )*), until the entire sound has loaded. From then on, it returns the tot number of bytes last transferred, until the next call to *loadSound ( )* .

For *Sound* objects, *getBytesLoaded ( )* is used with *getBytesTotal ( )* to determine (and display, normally) the load progre a downloading sound file. However, merely detecting the completion of a load operation does not require *getBytesLoadec* ; the *onLoad ( )* event handler is executed automatically when a sound finishes loading.

Example

The following code adds a new method, *preloadSound ( )* , to all *Sound* objects. It operates in the same way as *loadSound* except that it uses *getBytesLoaded ( )* to report load progress while the sound is loading. The *onLoad ( )* handler responds completion of the sound load operation, as usual. Note that as is, the code cannot load multiple sounds simultaneously because it does not queue preload requests. A queuing preloader is left as an exercise for the reader.

```
/*
 * Method: Sound.checkLoadProgress()
 *   Desc: Checks progress of a load(). Called only by preload().
 *         Reports progress by invoking a custom Sound.onBytesLoaded() handl
 */
Sound.prototype.checkLoadProgress = function () {
  // Check how many KB have loaded.
  var kbLoaded = Math.floor(this.getBytesLoaded()/1024);
  // Check the size of the file loading, in KB.
  var kbTotal = Math.floor(this.getBytesTotal()/1024);
  // Calculate the percent complete.
  var percentDone = isNaN(Math.floor(kbLoaded/kbTotal * 100)) ?
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

```
                    0 : Math.floor(kbLoaded/kbTotal * 100);
  // Execute onBytesLoaded(), which typically would display load progress.
  this.onBytesLoaded(this.getBytesLoaded(), this.getBytesTotal(),
                     kbLoaded, kbTotal, percentDone);
  // If the amount loaded is equal to the total file size,
  // the file is finished loading. However, we must ensure
  // that getBytesLoaded() returns a value greater than 0
  // because both getBytesLoaded() and getBytesTotal()
  // return 0 until the first bit of data arrives.
  if (this.getBytesLoaded() > 5
      && this.getBytesLoaded() =  = this.getBytesTotal()) {
    // ...stop calling checkLoadProgress().
    this.clearLoadCheck();
  }
}

/*
 *  Method: Sound.clearLoadCheck()
 *   Desc: Cancels an interval calling checkLoadProgress().
 */
Sound.prototype.clearLoadCheck = function () {
  // If there is a load interval for this Sound object...
  if (this.loaderID) {
    // ...stop calling it.
    clearInterval(this.loaderID);
  }
}

/*
 * Method: Sound.preloadSound()
 *   Desc: Begins the download of a sound file and executes
 *         checkLoadProgress() every 200 ms until the file has loaded.
 */
Sound.prototype.preloadSound = function (url, isStreaming) {
  // Always stop any previous preload before proceeding.
  this.clearLoadCheck();

  // Load the Sound file.
  this.loadSound(url, isStreaming);
  // Call checkLoadProgress() every 200 milliseconds.
  this.loaderID = setInterval(this, "checkLoadProgress", 200);
}

// Usage:
// Create a Sound object.
music = new Sound();
// Assign an onLoad() callback.
music.onLoad = function (success) {
  // If the load was successful...
  if (success) {
    // ...use the new sound.
    trace("done loading!");
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

```
    this.start();
  } else {
    // ...otherwise, perform failure operations and
    // cancel the preload interval (REQUIRED!).
    trace("load failed");
    this.clearLoadCheck();
  }
}
// Assign an onBytesLoaded() callback. This is where each individual
// Sound object can decide what to do with the preload information.
music.onBytesLoaded = function (bytesLoaded, bytesTotal,
                                kbLoaded, kbTotal, percentLoaded) {
  trace("Loading: " + kbLoaded +  " of " + kbTotal);
  trace(percentLoaded + " percent complete.");
}

// Load a streaming sound.
music.preloadSound("http://www.yoursite.com/sounds/song.mp3", true);
```

See Also

setInterval( ) , Sound.getBytesTotal ( ) , Sound.onLoad ( )

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Chapter 18. ActionScript Language Reference

Sound.getBytesTotal( ) Method Flash 6

the physical disk size of an external sound file, in bytes

```
soundObject.getBytesTotal()
```

Returns

An integer representing the size of the sound file, in bytes. Divide by 1024 to convert bytes to kilobytes (KB).

Description

The *getBytesTotal( )* method tells us the byte size of an external file being loaded into `soundObject` via *loadSound( )*. For a new *Sound* object, *getBytesTotal( )* returns `undefined`. Immediately after each call to *loadSound( )*, *getBytesTotal( )* also returns `undefined` until the size of the sound file being loaded is determined. After that, it returns the total byte size of the loading sound until the next call to *loadSound( )*.

For *Sound* objects, *getBytesTotal( )* is used together with *getBytesLoaded( )* to determine (and display, normally) the load progress of a downloading sound file. See *Sound.getBytesLoaded( )* for details and example code.

See Also

Sound.getBytesLoaded( ), *Sound.onLoad( )*

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Chapter 18. ActionScript Language Reference

Sound.getPan( ) Method Flash 5

retrieve the last pan value set

```
soundObject.getPan()
```

Returns

A number indicating the last value set by *setPan( )*. Usually in the range of -100 (left channel on, right channel off) to 100 (right channel on, left channel off). Default value is 0 (left and right channels in equal proportions).

Description

By adjusting the pan of a sound, you can create the illusion of a moving sound source. The *getPan( )* method is used to determine the current distribution of the left and right channels of the sounds controlled by `soundObject`. Normally, *getPan( )* is used in combination with *setPan( )* to adjust the current pan of a sound.

Example

Here we alter the pan of a sound by 20 towards the left channel:

```
mySound = new Sound();
mySound.setPan(mySound.getPan() - 20);
```

See Also

Sound.getTransform( ), Sound.setPan( )

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Chapter 18. ActionScript Language Reference

Sound.getTransform( ) Method Flash 5

determine the current distribution of the channels of a sound to the left and right speakers

```
soundObject.getTransform()
```

Returns

An anonymous object whose properties contain the channel percentage values for the sounds controlled by `soundObject`.

Description

The *getTransform( )* method returns an object with properties that tell us how the channels in the sounds controlled by `soundObject` are distributed to the left and right speakers. The properties of the returned object are `ll`, `lr`, `rl`, and `rr`, as described in the entry for the *Sound.setTransform( )* method.

See Also

Sound.getPan( ), *Sound.setTransform( )*

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Chapter 18. ActionScript Language Reference

Sound.getVolume( ) Method Flash 5

retrieve the current volume setting

```
soundObject.getVolume()
```

Returns

A number indicating the current volume as set by *setVolume( )*. Usually in the range of 0 (no volume) to 100 (default volume), but it can be higher.

Description

The *getVolume( )* method is used to determine the current volume of the sounds controlled by `soundObject`. Normally, *getVolume( )* is used in combination with *setVolume( )* to adjust the current volume of a sound.

Example

Here we reduce the volume of a sound by 20:

```
mySound = new Sound();
mySound.setVolume(mySound.getVolume() - 20);
```

See Also

Sound.setVolume( )

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Chapter 18. ActionScript Language Reference

Sound.id3 Property — Flash 6.0.40.0

access to the ID3 tags of an MP3 file — ***read-only***

soundObject.id3

Description

The id3 property is an object whose properties represent the ID3 tags of an MP3 sound file. MP3 files can be loaded intc *soundObject* from a URL, via *loadSound ( )* , or from the movie's Library via *attachSound ( )* . The available id3 o properties are listed in Table 18-18. Note that Flash Player 6 supports ID3v1 and ID3v1.1 tags only.

Table 18-18. Sound.id3 object properties

Property	Description
album	The name of the album from which the song originates.
artist	The performer of the song.
comment	Arbitrary notes about the song.
genre	An integer that maps to a description of the song's musical style. For details, see *Usage* .
songname	The title of the song.
track	The number of the song on its album. (ID3v1.1 only)
year	The year the song was produced.

An MP3 file must be loaded in its entirety before its ID3 tags are available (ID3v1 and ID3v1.1 tags are stored at the end MP3 file). If the MP3 file does not contain valid ID3 tags in a version supported by the Flash Player, the id3 object's properties are undefined .

Usage

The ID3 "standard" is not an official specification maintained by a recognized standards body. Rather, it is an agreed-upor system for embedding metadata in an MP3 file; this system was created by a single programmer, Eric Kemp (who also assembled the first list of 80 genre names). Genre names above 80 are not formalized, nor is the entire genre system considered adequate by most music professionals. For one list of genres, see:

http://www.mp3dev.org/mp3/doc/html/id3.html

For more information on the ID3 format, see http://www.id3.org/.

Bugs

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

The id3 properties are not immediately available from *Sound.onLoad ( )* . Use *setInterval ( )* to poll for a valid durati before accessing the id3 object, as shown in the following Example. (The id3 properties themselves will be undefin no ID3 tags are found, so they cannot be polled reliably.)

Example

The following code loads an MP3 file and displays its ID3 tags once it has finished loading:

```
// REQUIRES FLASH PLAYER 6.0.40.0 OR HIGHER
// Create a text field in which to display the ID3 information.
this.createTextField("output_txt", 1, 100, 100, 400, 300);
output_txt.border = true;

// Create a Sound object.
song = new Sound();
// When the sound loads, display the ID3 information.
song.onLoad = function (success) {
  // Store a reference to the current sound.
  var theSound = this;

  // Create a function that displays the ID3 info once sound duration is val
  function pollForDuration () {
    if (theSound.duration > 0) {
      // Display ID3 information.
      output_txt.text += "Title:  " + theSound.id3.songname + "\n";
      output_txt.text += "Artist: " + theSound.id3.artist + "\n";
      output_txt.text += "Album:  " + theSound.id3.album + "\n";
      output_txt.text += "Year:   " + theSound.id3.year + "\n";
      output_txt.text += "Track:  " + theSound.id3.track + "\n";
      output_txt.text += "Genre:  " + theSound.id3.genre + "\n";
      output_txt.text += "Notes:  " + theSound.id3.comment + "\n";

      // Play the sound.
      theSound.start();

      // Stop polling for a valid duration. This is required due to a
      // bug: the id3 property is not immediately available from Sound.onLoa
      clearInterval(intID);
    }
  }

  // If the sound loaded properly...
  if (success =  = true) {
    // ...repeatedly check for a valid duration.
    var intID = setInterval(pollForDuration, 100);
  }
}

// Load the MP3 file
song.loadSound("track1.mp3", false);
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

See Also

`capabilities.hasMP3` , *Sound.loadSound ( )* , *Sound.onLoad ( )*

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Chapter 18. ActionScript Language Reference

Sound.loadSound( ) Method Flash 6

download an external sound file into the Player

```
soundObject.loadSound(url, isStreaming)
```

Arguments

url

A string specifying the network or local path of the sound file to load.

isStreaming

A Boolean specifying the type of sound to load: either streaming (`true`) or event (`false`).

Description

The *loadSound ( )* method loads an external sound file from `url` . In Flash 6, only MP3 files can be loaded via *loadSound ( )* . If `isStreaming` is `true` , the loaded sound will behave as a streaming sound and will automatically begin playback as soon as the required buffer has loaded (see the global `_soundbuftime` property). If `isStreaming` is `false` , the loaded sound will behave as an event sound and will not begin playing automatically. To play a loaded event sound, use *Sound.start ( )* after the sound has fully loaded (see the following Example). To stop the playback of a loaded sound (either streaming or event), use *Sound.stop ( )* . In versions of the Flash Player prior to 6.0.40.0, a streaming sound cannot be played via *Sound.start ( )* . See that method's entry for more information.

A loading sound file's download progress can be monitored via *getBytesLoaded( )* (see that method's entry for sample preloading code). When the sound finishes loading, *Sound.onLoad ( )* executes. If the sound file cannot be found at `url` , *loadSound ( )* fails silently and *Sound.onLoad ( )* executes with a `success` parameter set to `false` .

To load a sound from a movie's Library instead of from a file, use *attachSound( )* . Unlike attached sounds, multiple loaded sounds can be loaded into a single *Sound* object simultaneously. However, the `position` and `duration` properties and the *start ( )* method apply only to the most recently loaded sound. Hence, for best control over multiple sounds, each sound file should be loaded into its own *Sound* instance. When both attached and loaded sounds are present in a single *Sound* object, *start ( )* applies to the loaded sound, even if the attached sound is added after the loaded sound is loaded. For example:

```
// Load track1.mp3
bgMusic.loadSound("track1.mp3");
// Sometime after track1.mp3 loads, attach track2 using attachSound()
bgMusic.attachSound("track2");
// Play the sound...
bgMusic.start();  // Plays track1.mp3 (the loaded sound)
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Flash can play a maximum of eight individual sounds simultaneously. When loaded via *attachSound( )* , each sound needs its own *Sound* instance.

To determine whether the Flash Player being used supports MP3 audio playback, use `System.capabilities.hasMP3` .

Usage

Compressed MP3 files often contain minor imperfections (such as an initial delay) that affect looped playback in Flash. To compensate, a loaded event sound can be offset by a small amount, as follows:

```
loopSound.start(0.05, 3);
```

However, this technique does not account for any ending silence on the sound, so results may vary. For best loop performance, use ADPCM or MP3 compression directly in Flash and attach the loop sound from the Library using *attachSound( )* .

Example

To load a sound, we first create a *Sound* object, then call the *loadSound ( )* method. Often, it's also a good idea to respond to the completion of the sound's download by defining a callback function for *onLoad( )* , as shown in the following example. First, we load an event sound; notice that the *onLoad( )* handler must be defined before the *loadSound ( )* call:

```
// Create a Sound object
music = new Sound();
// Define an onLoad() callback function for our object
music.onLoad = function () {
  // Place code here to respond to sound load completion
  trace("Sound has fully loaded");
  // It's an event sound and it's fully loaded, so we can safely play it now
  this.start();
}
// Load an event sound
music.loadSound("http://www.somesite.com/song.mp3", false);
```

Now, we load the same sound, but as a streaming sound. It's not as important to define an *onLoad ( )* callback for a streaming sound because the sound starts playing automatically when it has been buffered sufficiently:

```
// Create a Sound object
music = new Sound();
// Load a stream sound
music.loadSound("http://www.somesite.com/song.mp3", true);
```

See Also

`capabilities.hasAudio` , *Sound.getBytesLoaded ( )* , *Sound.getBytesTotal ( )* , *Sound.onLoad ( )* , `_soundbuftime`

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Chapter 18. ActionScript Language Reference

Sound.onLoad( ) Event Handler Flash 6

occurs when an external sound finishes loading

```
soundObject.onLoad(success)
```

Arguments

success

A Boolean: `true` if the load succeeded; otherwise, `false`. If the sound file was not found in a call to *loadSound( )*, *onLoad( )* is executed with a `success` parameter of `false`.

Description

The *onLoad( )* event handler executes when a sound file has downloaded completely into `soundObject` due to an earlier call to *loadSound( )*. To use *onLoad( )*, we assign it a *callback* (i.e., a custom-made function) that performs the action we take when our sound loads properly and/or the action we take when the load attempt fails. For example:

```
// Create the Sound object
carSound = new Sound();
// Assign a callback
carSound.onLoad = function (success) {
  if (success) {
    // Respond to completed sound load operation here
  } else {
    // Execute load-failure code here
  }
}
```

If the loaded sound is an event sound, we can use *onLoad( )* to start playing it, as follows:

```
carSound = new Sound();
carSound.onLoad = function (success) {
  if (success) {
    this.start();
  } else {
    displayLoadError();
  }
}
carSound.loadSound("http://www.somesite.com/sounds/driving.mp3", false);
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

The *onLoad( )* callback handler should be defined before the call to *loadSound( )* is made. Notice that from the body of the *onLoad( )* handler we can refer to our sound object with the `this` keyword.

Before a sound has fully loaded, we can check its download progress with *getBytesLoaded( )*.

Bugs

The `duration`, `position`, and `id3` *Sound* properties are not initially available from within an *onLoad( )* handler. See those property entries for details and workarounds.

See Also

Sound.getBytesLoaded( ), *Sound.getBytesTotal( )*, *Sound.loadSound( )*, *Sound.onSoundComplete( )*

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Chapter 18. ActionScript Language Reference

Sound.onSoundComplete( ) Event Handler Flash 6

occurs when a sound finishes playing

soundObject.onSoundComplete()

Description

The *onSoundComplete( )* event handler executes when an attached or loaded sound finishes playing (i.e., when *soundObject.position = = soundObject.duration*). It can be used to create a playlist of sounds that play in sequence or to cue some animation that starts precisely when a sound ends. To use *onSoundComplete( )*, we assign it a callback function that performs the action we wish to take when our sound finishes playing. For example, the following code loads an "introduction" voiceover track. When the track is done, the menu at frame `chapter1` appears:

```
narrative = new Sound();
narrative.attachSound("introduction");
narrative.start();

narrative.onSoundComplete = function () {
  // Intro's done. Display the chapter1 menu.
  _level0.gotoAndStop("chapter1");
}
```

Flash checks ten times per frame tick to see if the *onSoundComplete( )* event should be triggered. Hence, theoretically, movies with higher frame rates have more accurate *onSoundComplete( )* events.

Example

The following code displays the playback progress of a sound in the Output window:

```
// Create a Sound object
s = new Sound();
// Attach and play a Library sound
s.attachSound("groove");
s.start();

// When the sound finishes, display "done!" in the Output window
s.onSoundComplete = function () {
  trace("done!");
}
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

```
// Every frame, check how much of the sound has played
this.onEnterFrame = function () {
  trace("Played: " + s.position + " of " + s.duration + " milliseconds");
}
```

See Also

Sound.duration, *Sound.onLoad( )*, *Sound*.position

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Sound.position Property

Flash 6

the length of time the sound has played, in milliseconds ***read-only***

soundObject.position

Description

The position property indicates how long an attached or loaded sound has been playing, in milliseconds. If multiple s loaded into a single *soundObject* , the position property reflects the first loaded sound. See the following Bugs The position property can be used to cue animations that synch directly to the playback of a sound. For example, this an explosion animation after a sound has played for five seconds:

```
// Create the sound
introMusic = new Sound();
// Attach the Library sound exported as "song"
introMusic.attachSound("song");

// Every frame, check if the sound has played for 5 seconds
this.onEnterFrame = function () {
  // If the sound has played for 5 seconds...
  if (introMusic.position > 5000) {
    // Show the explosion
    this.gotoAndPlay("explosion_animation");
    // Stop checking the sound position
    delete this.onEnterFrame;
  }
}

// Start playing the sound
introMusic.start();
```

Used with the duration property, position can also tell us what percentage of a sound has played so far (see the fc Example). However, for looped sounds, position resets to 0 with each loop. Hence, to determine the percentage of a l track that has played, the position of each loop must be tallied and compared to the sound's duration multiplied b number of loops.

Before a sound plays, its position property returns 0, but after a sound completes, position is not reset to 0; inste equals the sound's duration .

The position property is read-only and cannot be used to play a sound at a certain position. Use *Sound.start ( )* with a *offset* value to start a sound at a specified time offset.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Bugs

In Flash 6, the `position` property is not reset for each sound loaded into a single *Sound* instance. For example, if the i `theMusic` loads and completely plays *song1.mp3* and then loads *song2.mp3* , its `position` property will be equal to *song1.mp3* 's total duration. To avoid this problem, load each sound into a unique *Sound* instance. Where possible, delete instances to save memory.

In Flash 6, while a streaming sound is loading, the `position` property can be inaccurate (on the order of milliseconds) nonstreaming sounds when `position` accuracy is critical (for example, when synching multiple sounds).

In Flash 6, the `position` property is not initially available within *onLoad ( )* . Use *setInterval ( )* to poll for a valid `pos` as follows:

```
song = new Sound();
song.onLoad = function (success) {
  // Because of the bug, no position displays
  trace("Initial Position: " + this.position);

  // Store a reference to the current sound
  var theSound = this;

  // Create a function that checks if position exists
  function pollForPosition () {
    if (theSound.position >= 0) {
      // Show the position, now that it's available
      trace("Actual position: " + theSound.position);
      // Stop polling for a valid position
      clearInterval(intID);
    }
  }

  // If the sound loaded properly...
  if (success =  = true) {
    // Play the sound
    theSound.start();
    // Repeatedly check for a valid position
    var intID = setInterval(pollForPosition, 10);
  }
}

// Load the MP3 file
song.loadSound("track1.mp3", false);
```

Example

The following code adds to all sounds a *startFadeIn ( )* method that increases the volume of a sound proportionate to the a played. As is, the method works with attached sounds only, because it lacks the preloader code required for applying the e loaded sounds:

```
/*
 * Method: Sound.startFadeIn()
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

```
 *   Desc: Fades a sound in from 0-100% volume.
 * Params: fadeDuration.  The length, in milliseconds, of the fade in.
 */
Sound.prototype.startFadeIn = function (fadeDuration) {
  this.setVolume(0);
  this.fadeDuration = (fadeDuration =  = undefined) ? this.duration : fadeDu
  this.volInterval = setInterval(this, "fadeIn", 10);
}

/*
 * Method: Sound.fadeIn()
 *   Desc: Internal method that performs a sound fade. Called by startFadeIn
 */
Sound.prototype.fadeIn = function () {
  this.vol = Math.floor(this.position / this.fadeDuration * 100);
  this.setVolume(this.vol);
  // Stop calling fadeIn() if the sound has finished playing
  if (this.fadeDuration < this.position) {
    clearInterval(this.volInterval);
  }
  trace("volume: " + this.vol);
}

// Usage:
// Create a new Sound object.
var music = new Sound();

// Attach a sound from the Library and play it
music.attachSound("song");
music.start();

// Invoke the startFadeIn() method with a 2 second fade (2000 milliseconds)
music.startFadeIn(2000);
```

See Also

Sound.attachSound ( ) , Sound .duration , *Sound.loadSound ( )*

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Chapter 18. ActionScript Language Reference

Sound.setPan( ) Method Flash 5

set the balance of a sound's left and right channels

soundObject.setPan(*pan*)

Arguments

pan

A number between -100 (left) and 100 (right), indicating the distribution between the left and right speakers for sounds controlled by *soundObject* . If the *pan* supplied is greater than 100, the actual value assigned is 200 - *pan* . If the *pan* supplied is less than -100, the actual value assigned is -200 - *pan* .

Description

The *setPan ( )* method dictates the balance of the right and left channels of the sounds controlled by *soundObject* . By adjusting the pan over time, we can cause a sound to move from one speaker to the other (known as *panning*).

To play the sounds controlled by *soundObject* in the left speaker only, use a *pan* of -100. To play the sounds controlled by *soundObject* in the right speaker only, use a *pan* of 100. To balance the two channels evenly, use a *pan* of 0.

Note that *setPan ( )* affects all the sounds controlled by *soundObject* . If *soundObject* is a Player-wide sound, *setPan ( )* affects all the sounds in a movie. If *soundObject* is tied to a clip or a main timeline, *setPan( )* affects all the sounds in that clip or timeline and all the clips it contains.

The effects of *setPan ( )* can be changed only by another call to *setPan ( )* . A *setPan ( )* assignment affects all future sounds controlled by *soundObject* , even if *soundObject* is deleted.

Example

The following code adds a new method, *doPanEffect ( )* , to all movie clips, which causes sounds in a movie clip to pan endlessly between the left and right speakers:

```
// Method: MovieClip.doPanEffect()
MovieClip.prototype.doPanEffect = function () {
  var panEffect = new Sound(this);
  var panDirection = "right";
  var panIncrement = 10;

  this.onEnterFrame = function () {
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

```
    if (panDirection =  = "right") {
      newPan = panEffect.getPan() + panIncrement;
      if (newPan > 100) {
        panDirection = "left";
        panEffect.setPan(panEffect.getPan() - panIncrement);
      } else {
        panEffect.setPan(newPan);
      }
    } else {
      newPan = panEffect.getPan() - panIncrement;
      if (newPan < -100) {
        panDirection = "right";
        panEffect.setPan(panEffect.getPan() + panIncrement);
      } else {
        panEffect.setPan(newPan);
      }
    }
  }
}

// Usage:
theSoundClip_mc.doPanEffect();
```

The following example causes sounds in `soundClip_mc` to react to the mouse. Assuming a Stage width and height of 550 and 400, the sounds pan left and right with the mouse's horizontal position and increase or decrease in volume with the mouse's vertical position. To try out the effect, create a new movie clip instance named `soundClip_mc` on the main timeline and attach the following code to frame 1 of the main timeline. Then either place sounds inside `soundClip_mc` or export a sound from the Library with an export identifier of "bgMusic".

```
// Create a new Sound object and attach the sound bgMusic to it
soundClip_mc.volControl = new Sound(soundClip_mc);
soundClip_mc.volControl.attachSound("bgMusic");
soundClip_mc.volControl.start(0, 999);  // Play and loop the sound

// With every frame, set volume and pan
soundClip_mc.onEnterFrame = function () {
  // Measure the mouse's horizontal location, then set the pan accordingly
  var mouseX = (_root._xmouse / 550) * 200;
  this.volControl.setPan(mouseX - 100);
  // Measure the mouse's vertical location, then set the volume accordingly
  var mouseY = (_root._ymouse / 400) * 300;
  this.volControl.setVolume(300 - mouseY);
}
```

See Also

Sound.getPan ( ) , Sound.setTransform ( ) , Sound.setVolume ( )

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Chapter 18. ActionScript Language Reference

Sound.setTransform( ) Method Flash 5

distribute the left and right channels between the left and right speakers

soundObject.setTransform(*transformObject*)

Arguments

transformObject

A user-defined object that specifies new channel settings as a series of reserved properties.

Description

The *setTransform( )* method gives us precise control over how the channels in a sound are output to the left and right speakers. In principle, *setTransform( )* is not unlike *setPan( )*, but it provides more detailed sound control over stereo sounds.

A stereo sound is a combination of two distinct sounds—the *left channel* and the *right channel*—which normally are sent separately to the left and right speakers. However, using *setTransform( )*, we can dictate how much of each channel is broadcast in each speaker. For example, we can tell Flash to play half of the left channel in the left speaker, none of the left channel in the right speaker, and all of the right channel in both speakers. Or we can tell Flash to play all of the left and right channels in the left speaker only.

To use *setTransform( )*, we must first create an object with a series of predefined properties. The properties express how to distribute the left and right channels of a stereo sound between the left and right speakers, as described in Table 18-19.

Table 18-19. Properties of a transformObject

Property name	Property value	Property description
`ll`	0 to 100	The percentage of the left channel to play in the left speaker
`lr`	0 to 100	The percentage of the right channel to play in the left speaker
`rl`	0 to 100	The percentage of the left channel to play in the right speaker
`rr`	0 to 100	The percentage of the right channel to play in the right speaker

Once we have created an object with the properties described in Table 18-19, we pass that object to the *setTransform( )* method of our *Sound* object. The values of the properties on our `transformObject` become the new channel output percentages for the sounds controlled by `soundObject`.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

To examine the current percentages of a particular *Sound* object, we use the *Sound.getTransform( )* method.

Example

```
// Create a new Sound object
mySound = new Sound();

// Create a new generic object to use with setTransform( )
transformer = new Object();

// Set the properties of the transform object
transformer.ll = 0;      // None of left channel in left speaker
transformer.lr = 0;      // None of right channel in left speaker
transformer.rl = 0;      // None of left channel in right speaker
transformer.rr = 100;    // All of right channel in right speaker

// Apply the new channel distribution by passing the transform
// object to the setTransform( ) method
mySound.setTransform(transformer);
```

See Also

Sound.getTransform( ), Sound.setPan( ), Sound.setVolume( )

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Chapter 18. ActionScript Language Reference

Sound.setVolume( ) Method Flash 5

set the volume of sounds controlled by a Sound object

```
soundObject.setVolume(volume)
```

Arguments

volume

A number indicating the loudness of the sound controlled by `soundObject` , where 0 is no volume (mute). The larger `volume` 's absolute value (regardless of whether `volume` is positive or negative), the louder the sounds controlled by `soundObject` will be. For example, -50 is the same `volume` as 50. The default value for `volume` is 100.

Description

The *setVolume ( )* method makes the sounds controlled by `soundObject` louder or softer. To entirely mute a sound, use a `volume` of 0. To make a sound louder, increase `volume` 's absolute value. Values in the range of 100-200 are generally quite loud, but there is no predefined maximum.

Note that *setVolume ( )* affects all the sounds controlled by `soundObject` . If `soundObject` is a global sound, *setVolume ( )* affects all the sounds in a movie. If `soundObject` is tied to a clip or a main timeline, *setVolume ( )* affects all the sounds in that clip or timeline.

The volume set by *setVolume( )* remains in effect until it is overridden by another *setVolume ( )* call. A *setVolume ( )* assignment affects all future sounds controlled by `soundObject` , even if `soundObject` is deleted.

Example

This first example simply sets the volume of a movie clip named `theClip_mc` :

```
var mySound = new Sound(theClip_mc);
mySound.setVolume (65);
```

The following example shows how to make buttons that adjust a movie's volume level:

```
var globalSound = new Sound();
var maxVolume = 200;
var minVolume = 0;
var volumeIncrement = 20;
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

```
volumeUp_btn.onRelease = function () {
  globalSound.setVolume(Math.min(globalSound.getVolume() + volumeIncrement,
                                 maxVolume));
}

volumeDown_btn.onRelease = function () {
  globalSound.setVolume(Math.max(globalSound.getVolume() - volumeIncrement,
                                 minVolume));
}
```

See Also

Sound.getVolume ( ) , Sound.stop ( ) , Sound.stopAllSounds ( )

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Sound.start( ) Method Flash 5

begin playing an attached or loaded event sound

```
soundObject.start(secondOffset, loops)
```

Arguments

secondOffset

A floating-point number indicating the time, in seconds, at which to start playing the sound attached to `soundObject` (often called an *entry point*). For example, a `secondOffset` of 1 starts playback one secor after the sound's actual beginning, as defined in the Library. The default is 0. There is no provision for an *exit poi* (the time at which to stop playing the sound). The sound plays until its end unless it is stopped manually. See the `position` property for code that stops a sound after `n` milliseconds.

loops

A positive integer indicating how many times to play the sound attached to `soundObject` . To play the sound once, use 1 (the default); to play the sound twice in succession, use 2; and so on. The portion of the sound from `secondOffset` is repeated to its end the number of times specified by `loops` . Applies to nonstreaming sounds only.

Description

The *start ( )* method is used to play programmatically defined sounds that were added to `soundObject` via *attachSour)* or *loadSound ( )* . The *start ( )* method does not play all the sounds in a clip or movie; it plays only the sound most recen attached to `soundObject` via *attachSound ( )* or *loadSound ( )* , and it always plays a loaded sound in favor of an attached sound if both are present in the same *Sound* object.

To play only a portion of the sound attached to `soundObject` , use the `secondOffset` argument. To play the sou attached to `soundObject` repeatedly, use the `loops` argument. Normally, multiple calls to *soundObject.start( )* play multiple instances of the same sound (rather than restarting the sound), except when the sound is a streaming loaded sounc in which case the sound begins playing at the specified `secondOffset` . In general, sounds should be stopped via *sto)* before being restarted.

The *start ( )* method works with loaded streaming sounds in Flash Player 6.0.40.0 and later only. However, the `loops` parameter is not available for streaming sounds in any version of the Player.

Example

```
// Create a new Sound object
boink = new Sound();
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

```
// Attach a sound exported as boink to the Sound object
boink.attachSound("boink");

// Play all of boink (soundOffset defaults to 0)
boink.start();

// Play only a portion of boink, starting 0.5 seconds in (loops defaults to
boink.start(.5);

// Play boink three times from beginning to end
boink.start(0, 3);
```

See Also

Sound.loadSound ( ) , *Sound* .position , *Sound.stop ( )*

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Chapter 18. ActionScript Language Reference

Sound.stop( ) Method Flash 5

silence all sounds or a specified attached or loaded sound

```
soundObject.stop()
soundObject.stop(linkageIdentifier)
```

Arguments

linkageIdentifier

The name of any sound attached to any *Sound* object with the same `target` as `soundObject`. Linkage identifiers are specified in the Library under Options Linkage.

Description

When invoked without a `linkageIdentifier`, *stop( )* silences all sounds controlled by `soundObject`; if `soundObject` is a global sound, *stop( )* silences all sounds in a movie; if `soundObject` was created with a `target` parameter, *stop( )* silences all sounds in `target`.

When invoked with a `linkageIdentifier`, *stop( )* silences only the specific sound named by `linkageIdentifier`. In that case, `linkageIdentifier` must be a sound that was attached to a *Sound* object via *attachSound( )*. However, the sound to stop need not be attached to `soundObject` itself. It can be attached to *any Sound* object that shares the same `target` as `soundObject`. Or, if `soundObject` was created with no `target` (i.e., is a global *Sound* object), the sound to stop can be attached to any other global *Sound* object.

Example

```
// Create a global Sound object
mySound = new Sound();

// Attach the sound doorbell to the object
mySound.attachSound("doorbell");

// Stop all sounds in the movie
mySound.stop();

// Play doorbell
mySound.start();

// Stop just doorbell
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

```
mySound.stop("doorbell");

// Create another global Sound object
myOtherSound = new Sound();

// Attach a doorknock sound to the object
myOtherSound.attachSound("doorknock");

// Play doorknock
myOtherSound.start();

// Now stop the doorknock through mySound, not myOtherSound.
// This works because the two Sound objects have the same target.
mySound.stop("doorknock");
```

See Also

Sound.attachSound( ), Sound.start( ), stopAllSounds( )

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Chapter 18. ActionScript Language Reference

_soundbuftime Global Property — Flash 4

length of a streaming sound, in seconds, to preload — ***read/write***

`_soundbuftime`

Description

The `_soundbuftime` property is an integer specifying the number of seconds of a streamed sound to preload before playing the sound (the default is 5 seconds). It applies to both timeline sounds and sounds loaded at runtime via *Sound.loadSound( )*.

Flash synchronizes movie playback with streaming sounds to ensure that, say, a cartoon character's lips match an accompanying soundtrack. Animations will pause until `_soundbuftime` seconds of streaming sound are downloaded; therefore, a large setting (long delay) can cause excessive startup times on slower connections. Because network streaming may be slow or briefly interrupted, a short `_soundbuftime` setting can cause sound to skip (i.e., if enough sound data wasn't buffered, sounds may be interrupted temporarily). The ideal setting will vary from movie to movie based on the complexity of the graphics, the quality settings of the sound, and the bandwidth of the end user's Internet connection. The default setting (5 seconds) usually works well, but experimentation may be required to find the best setting for individual cases. The streaming buffer time can be changed during playback, but it is a global property and cannot be set separately for individual sounds.

Example

```
_soundbuftime = 10;  // Buffer 10 seconds of audio
```

See Also

The *Sound* object, *stopAllSounds( )*

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Chapter 18. ActionScript Language Reference

Stage Object — Flash 6

access to a movie's size, scale settings, and alignment

Properties

align

Placement of the movie in the Flash Player.

height

Pixel height of the available Flash Player display area or the author-time document height, depending on the value of `scaleMode`.

scaleMode

Dictates how the movie is sized, relative to the Player.

showMenu

Boolean; determines the items displayed in the Flash Player's contextual menu.

width

Pixel width of the available Flash Player display area or the author-time document width, depending on the value of `scaleMode`.

Methods

addListener( )

Registers an object to receive *onResize( )* events.

removeListener( )

Cancels event notices for the specified listener.

Listener Events

onResize( )

Occurs when movie is resized.

Description

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

The *Stage* object represents the characteristics of the Flash Player's runtime display area. It tells us the current dimensions of the movie, notifies us when the movie changes size, and provides control over global movie settings, such as which options appear in the Player's contextual menu (brought up via right-click on Windows or Ctrl-click on Macintosh). Mostly, the *Stage* object is used to adjust the visual layout of a movie when the browser or Standalone Player is resized.

Note that *Stage* is a predefined object, not a class. Its properties and methods are accessed directly, as in:

```
trace(Stage.width);  // Displays the width of the movie
```

See Also

MovieClip`._height`, *MovieClip*`._width`

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Chapter 18. ActionScript Language Reference

Stage.addListener( ) Method Flash 6

registers an object to receive onResize( ) events

```
Stage.addListener(listener)
```

Arguments

listener

An object with an *onResize( )* method defined.

Description

The *addListener( )* method adds `listener` to the list of objects notified when *Stage*'s *onResize( )* event occurs. The `listener` argument is an object of any class that defines a method by the name *onResize*; the method is invoked when the corresponding event occurs for *Stage*. Instances of built-in classes, such as *TextField*, *Button*, or *MovieClip*, can all respond to *Stage.onResize( )*. To stop a listener object from receiving events, use *Stage.removeListener( )*.

Multiple objects can respond independently to the *onResize( )* listener event. When multiple listeners are registered to *Stage*, their methods are invoked in the order the listeners were added.

For general information on how listeners work, see Chapter 10.

Example

The following code assigns an *onResize( )* method to the `ball_mc` movie clip and registers the clip to respond to the *onResize( )* event. Note that `Stage.scaleMode` must be set to "noScale" in order for the *onResize( )* event to be triggered:

```
Stage.align = "LT";
Stage.scaleMode = "noScale";
ball_mc.onResize = function () {
  // Place ball_mc in the center of the Stage (assumes that
  // ball_mc's content is centered on its registration point).
  // Inside the method, 'this' refers to ball_mc.
  this._x = Stage.width/2;
  this._y = Stage.height/2;
}
Stage.addListener(ball_mc);
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

See Also

Stage.onResize( ), *Stage.removeListener*( ), `Stage.scaleMode`; Chapter 10

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Chapter 18. ActionScript Language Reference

Stage.align Property Flash 6

alignment of the movie within the Flash Player ***read/write***

```
Stage.align
```

Description

When a movie is embedded in a web page, the HTML `OBJECT` and `EMBED` tags can specify dimensions that do not match those of the Flash document. When the movie is displayed in the Standalone Player, the Player can be resized by the user. In both cases, the string `align` property dictates the positioning of the movie, relative to the space available to the Flash Player. By default, `align` is the empty string, indicating that the movie should be centered horizontally and vertically in the Player. Table 18-20 lists the available alignment settings:

Table 18-20. Stage.align property values

align property	Description	Horizontal alignment	Vertical alignment
"" (empty string)	centered	center	center
"B"	bottom, center	center	bottom
"T"	top, center	center	top
"L"	left side, center	left	center
"LB"	left, bottom corner	left	bottom
"LT"	left, top corner	left	top
"R"	right side, center	right	center
"RB"	right, bottom corner	right	bottom
"RT"	right, top corner	right	top

The effect of `align` is most obvious (and most useful) when `scaleMode` is set to "noScale", in which case the movie does not change size in the Player — it changes placement only.

When the Player is resized, the *.swf* document it contains is automatically repositioned according to `align`. To force a custom positioning for elements in a movie at resize time, use the *onResize( )* listener event.

When a movie is embedded in a web page, the `OBJECT` or `EMBED` tag can also set the value of `align`; any assignment in ActionScript overrides the HTML setting.

Usage

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript corrects any imprecise assignments of `align` to match the closest valid setting. For example, assigning `"tl"` results in `"LT"`, and assigning `"jjjt"` results in `"T"`. For code clarity, however, the documented settings should always be used.

Example

The following code pins the movie to the top-left corner of the Player:

```
Stage.scaleMode = "noScale";
Stage.align = "LT";
```

See Also

Stage.onResize( ), `Stage.scaleMode`

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Chapter 18. ActionScript Language Reference

Stage.height Property — Flash 6

pixel height of the available Flash Player display area or author-time document — ***read-only***

```
Stage.height
```

Description

The integer `height` property indicates the pixel height of the movie, depending on the setting of `Stage.scaleMode`. When `scaleMode` is set to "noScale", `height` reports the size of the Flash Player's available display area—either the Standalone Player window's dimensions or the current size of the `OBJECT/EMBED` region in the browser. When `scaleMode` is not set to "noScale", `height` reports the size of the Flash movie as specified at authoring time via Modify Document Dimensions.

In Test Movie mode, `scaleMode` defaults to "noScale", so `height` may not match the dimensions set at authoring time. In a browser or the Standalone Player, `scaleMode` defaults to "showAll", so `height` matches the dimensions set at authoring time unless `scaleMode` is explicitly set to "noScale".

To receive notification when `Stage.height` changes, add listeners for the *Stage.onResize( )* listener event.

Usage

Don't confuse `Stage.height` with `_root._height`. The former is the height of the available Player space or the specified document dimensions, whereas the latter is the height of all objects on the main movie timeline.

Bugs

When `scaleMode` is "noScale" and `height` is accessed on the first few frames of a movie, an incorrect value may be reported. For safety, wait several frames before accessing `height`, or, better yet, access it only within an *onResize( )* listener.

In Test Movie mode, the `height` of the movie is reported as four pixels less than actual size.

Example

The following code creates a text field and centers it on stage:

```
Stage.scaleMode = "noScale";
Stage.align = "LT";
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

```
this.createTextField("theField_txt", 1, 0, 0, 200, 20);
theField_txt.text = "Derek got a haircut.";
theField_txt._x = Stage.width/2 - theField_txt._width/2;
theField_txt._y = Stage.height/2 - theField_txt._height/2;
```

See Also

MovieClip.createTextField( ), *MovieClip*`._height`, *Stage.addListener( )*, *Stage.onResize( )*, `Stage.width`, the *TextField* class

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Chapter 18. ActionScript Language Reference

Stage.onResize( ) Listener Event Flash 6

invoked when the movie is resized

listenerObject.onResize( )

Description

The *onResize ( )* event is triggered when `Stage.scaleMode` is set to "noScale" and the Player dimensions change. T Player dimensions change when either:

- The Standalone Player is resized by the user or in response to an *fscommand ("fullscreen", "true")* call.
- The user resizes a browser window containing a movie embedded using percentage values for the `OBJECT/EMBE` tag's `HEIGHT` or `WIDTH` attributes.

The *onResize ( )* event lets us dynamically reposition elements in a movie, relative to the currently available space. We ca use this ability to create "stretchy" layouts that behave like percentage-width tables in web pages created with HTML. A movie in a browser window triggers resize events only if its HTML `HEIGHT` or `WIDTH` is set to a percentage value. For example:

```
<OBJECT classid="clsid:D27CDB6E-AE6D-11cf-96B8-444553540000"
 codebase="http://download.macromedia.com/pub/shockwave/cabs/flash/swflash.c
 WIDTH="100%"
 HEIGHT="75">
 <PARAM NAME=movie VALUE="theMovie.swf">

 <EMBED src="theMovie.swf"
  WIDTH="100%"
  HEIGHT="75"
  TYPE="application/x-shockwave-flash"
  PLUGINSPAGE="http://www.macromedia.com/go/getflashplayer">
 </EMBED>
</OBJECT>
```

Notice that one dimension can be set to a fixed pixel size (`HEIGHT="75"`) while the other is a percentage (`WIDTH="100%"`). When a movie first loads into a browser, *onResize ( )* is not triggered, even if percentage dimensio are specified. We must trigger the movie's initial layout code manually.

The following code registers an event listener that responds to *onResize ( )* . Notice the mandatory setting of `scaleMod` "noScale":

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

```
Stage.align = "T";              // Position the movie at top, center
Stage.scaleMode = "noScale";  // Prevent scaling (required!)

// Create the listener object
resizeListener = new Object();
resizeListener.onResize = function () {
  trace("New width: " + Stage.width);
};
Stage.addListener(resizeListener);
```

Any number of listener objects can respond to the *onResize ( )* event. For example, in dynamic layouts, each visual object text field, button, or movie clip) can be registered as a listener for *onResize ( )* so that each element can position itself independently when the movie changes size, as discussed in Chapter 10. See the following Example for code that position movie clip when the movie is resized.

To stop a listener from responding to the *onResize ( )* event, use *removeListener ( )* , as in:

```
Stage.removeListener(resizeListener);
```

Visual changes made to a movie inside an *onResize ( )* listener are rendered immediately, independent of any frame rate setting. No call to *updateAfterEvent ( )* is required.

Example

The following code pins the movie clip `logo_mc` to the top-right corner of the Player's display area even when the Stage resized:

```
Stage.align = "LT";
Stage.scaleMode = "noScale";
stageListener = new Object();
stageListener.onResize = function () {
  logo_mc._x = Stage.width - logo_mc._width;
  logo_mc._y = 0;
}
Stage.addListener(stageListener);
```

A more extensive sample of a resizable star field can be downloaded from:

http://www.moock.org/webdesign/flash/actionscript/resizableMovie/resizableMovie.zip

Bugs

When a call to *fscommand("fullscreen", "true")* triggers *onResize ( )* , the values of `Stage.width` and `Stage.heig` within the callback function are incorrect. To determine the actual size of the fullscreen Standalone Player, use *setInterval* to poll for the `width` and `height` , or check the screen size directly with `System.capabilities.screenResolutionX` and `System.capabilities.screenResolution`

See Also

Stage.addListener ( ) , `Stage.height` , *Stage.removeListener* ( ), `Stage.scaleMode` , `Stage.width` ; Chapte 10.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Stage.removeListener( ) Method Flash 6

cancels event notices for the specified listener

```
Stage.removeListener(listener)
```

Arguments

listener

An object that was previously registered as a listener for `Stage`.

Returns

A Boolean: `true` if *`listener`* was found and removed, `false` if *`listener`* was not registered as an event listener for `Stage`. The return value is useful for debugging code that removes listeners.

Description

The *removeListener( )* method stops *`listener`* from receiving `Stage` event notifications. It should be used only after *`listener`* has been registered as an event listener via *addListener( )*. The listener object is not deleted, but it can be removed and added repeatedly. To delete a listener, first remove it with *removeListener( )* and then delete it with the *delete* operator.

Example

You should remove event listeners when the task they perform is no longer required. For example, a text field might have two display settings in an application, "percentage width" (stretch the field to match the movie width) and "fixed width" (stay the same width). A listener could be added to, or removed from, the field, depending on the current mode, as follows:

```
// Set scaleMode to allow onResize( ) to operate.
Stage.align = "LT";
Stage.scaleMode = "noScale";

// Create the text field.
this.createTextField("body_txt", 1, 5, 0, 200, 20);
body_txt.border = true;
body_txt.type = "input";
body_txt.onResize = function () {
  this._width = Stage.width - 10;
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

```
};

// Mode-switching function. Adds and removes the listener dynamically.
function setEditMode (mode) {
  if (mode =  = "fixed") {
    Stage.removeListener(body_txt);
  } else if (mode =  = "percent") {
    Stage.addListener(body_txt);
  }
}

// Set the mode.
setEditMode("percent");
```

See Also

Stage.addListener( ), *Stage.onResize( )*; Chapter 10

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Chapter 18. ActionScript Language Reference

Stage.scaleMode Property — Flash 6; partially available through fscommand() since Flash 3

dictates how the movie is sized, relative to the Player — ***read/write***

```
Stage.scaleMode
```

Description

The string `scaleMode` property governs how the movie scales (resizes) when the Flash Player's available space does not match the movie's Stage dimensions set at authoring time via Modify Document Dimensions. The allowed settings for `scaleMode` are:

"exactFit"

Forces every edge of the movie to the edge of the Player, distorting the movie's aspect ratio, if necessary, to fit the available space.

"showAll"

Fits the width or height of the movie to the smaller of the width or height of the Player. The movie will not be cropped or skewed to fit the Player, but there will be borders around the movie (according to the current setting of `Stage.align`). By default, either the top and bottom or right and left of the movie will have a border.

"noBorder"

Fits the height or width of the movie to the larger of the width or height of the Player. When the aspect ratio of the movie does not match the aspect ratio of the Player, the movie will be cropped according to the current setting of `Stage.align`. By default, either the top and bottom or both sides will be cropped.

"noScale"

Preserves the movie's author-time size. When the Player is smaller than the movie, the movie is cropped; when the Player is larger than the movie, the excess space displays the movie's background color. Positioning of the movie is controlled by the `Stage.align` property.

The default `scaleMode` for the Standalone Player and web browser plugin is "showAll". The default `scaleMode` for Test Movie mode is "noScale".

The `scaleMode` property normally is used in concert with the `align` property and the *onResize( )* event for movies with dynamically positioned content.

When a movie is embedded in a web page, the `OBJECT` or `EMBED` tag can also set the value of `scaleMode`;

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

any assignment in ActionScript overrides the HTML setting.

Usage

In Flash Player versions 3 through 5, the equivalent of `Stage.scaleMode = "noScale"` can be achieved via *fscommand("allowscale", "false")*.

Example

```
// Distort the movie to fill the Player area
Stage.scaleMode = "exactFit";

// Prevent scaling completely, regardless of the size of the Player
Stage.scaleMode = "noScale";
```

See Also

Stage.addListener( ), `Stage.align`, *Stage.onResize( )*

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Chapter 18. ActionScript Language Reference

Stage.showMenu Property

Flash 6; available through fscommand( ) since Flash 3

Boolean; determines the items displayed in the Flash Player's contextual menu ***read/write***

`Stage.showMenu`

Description

The Boolean `showMenu` property indicates whether the Flash Player's contextual menu contains a full list of items (`true`) or a minimal list of items (`false`). It defaults to `true` but is often set to `false` to prevent users from manipulating the movie in undesirable ways, such as stopping it, rewinding it, or zooming it. The contextual menu is accessed using right-click under Windows or Ctrl-click on the Macintosh.

In Flash 6, the following menu items are available when `showMenu` is `true`:

Zoom In
Zoom Out
100%
Show All
Quality
Play
Loop
Rewind
Forward
Back
Settings...
Print...
About Macromedia Flash Player 6

The following items are available when `showMenu` is `false`:

Settings...
About Macromedia Flash Player 6

In both cases, the Settings option leads to the Settings dialog box, where the user can set the *SharedObject* disk space limit and configure her camera and microphone.

There is no built-in way to disable the Flash Player's contextual menu entirely. However, several third-party tools can remove the menu from the Standalone Player. See:

http://www.flashjester.com/software/jugglor/ and http://www.northcode.com/studio/

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

When a movie is embedded in a web page, the `OBJECT` or `EMBED` tag can also set the value of `showMenu`; any assignment in ActionScript overrides the HTML setting.

Usage

In Flash versions 3 through 5, `showMenu` must be set via *fscommand( )*, such as *fscommand ("showMenu", "false")*.

Example

```
// Reduces the contextual menu to minimal items
Stage.showMenu = false;
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Chapter 18. ActionScript Language Reference

Stage.width Property — Flash 6

pixel width of the available Flash Player display area or the author-time document — ***read-only***

```
Stage.width
```

Description

The integer `width` property indicates the pixel width of the movie, depending on the setting of `Stage.scaleMode`. When `scaleMode` is set to "noScale", `width` reports the size of the Flash Player's available display area—either the Standalone Player window's dimensions or the current size of the `OBJECT/EMBED` region in the browser. When `scaleMode` is not set to "noScale", `width` reports the size of the Flash movie as specified at authoring time via Modify Document Dimensions.

In Test Movie mode, `scaleMode` defaults to "noScale", so `width` may not match the dimensions set at authoring time. In a browser or the Standalone Player, `scaleMode` defaults to "showAll", so `width` matches the dimensions set at authoring time unless `scaleMode` is explicitly set to "noScale".

To receive notification when `Stage.width` changes, add listeners for the *Stage.onResize( )* listener event.

Usage

Don't confuse `Stage.width` with `_root._width`. The former is the width of the available Player space or the specified document dimensions, whereas the latter is the width of all objects on the main movie timeline.

Bugs

When `scaleMode` is "noScale" and `width` is accessed on the first few frames of a movie, an incorrect value may be reported. For safety, wait several frames before accessing `width`, or, better yet, access it only within an *onResize( )* listener.

In Test Movie mode, the `width` of the movie is reported as four pixels less than actual size.

Example

The following code uses *MovieClip* drawing methods to draw a line across the entire width of the Stage, 30 pixels below its top border:

```
Stage.scaleMode = "noScale";
Stage.align = "LT";
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

```
this.lineStyle(1);
this.moveTo(0, 30);
this.lineTo(Stage.width, 30);
```

See Also

MovieClip._width, *Stage.addListener( )*, Stage.height, *Stage.onResize( )*

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Chapter 18. ActionScript Language Reference

startDrag( ) Global Function — Flash 4

make a movie or movie clip follow the mouse pointer

```
startDrag(target)
startDrag(target, lockCenter)
startDrag(target, lockCenter, left, top, right, bottom)
```

Arguments

target

A string or reference indicating the path to the movie or movie clip instance that should follow the mouse pointer (references to movie clips are converted to paths when used in a string context).

lockCenter

A Boolean indicating whether `target`'s registration point should be centered under the mouse pointer (`true`) or dragged relative to its original location (`false`).

left

A number specifying the minimum x-coordinate to the left of which `target`'s registration point cannot be dragged.

top

A number specifying the minimum y-coordinate above which `target`'s registration point cannot be dragged.

right

A number specifying the maximum x-coordinate to the right of which `target`'s registration point cannot be dragged.

bottom

A number specifying the maximum y-coordinate below which `target`'s registration point cannot be dragged.

Description

The *startDrag( )* function causes `target` to follow the mouse pointer around in the Player visually (known as *dragging* the clip). The movement of a dragged clip can be constrained to a bounding box whose coordinates are provided as arguments to the *startDrag( )* function. Bounding box coordinates are given relative to the canvas on which `target` resides. If that canvas is the main movie Stage, then (0,0) is the top-left corner of the Stage. If that canvas is a movie clip, then (0,0) is the registration point of the clip's canvas. Note that Flash's coordinate system

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

reverses the Cartesian Y-axis; y values *increase* toward the bottom of the screen and *decrease* toward the top of the screen. Negative y values are *above* the origin (i.e., above the X-axis).

Dragging can be stopped at any time via the *stopDrag( )* function. Only one movie clip or movie can be dragged at a time, so issuing a *startDrag( )* function on a new `target` automatically cancels any drag operation already in progress. That said, when a movie or movie clip is dragged, all the movie clips it contains are dragged along with it.

Example

```
// Drag ball, limiting its movement to the upper-left corner of the Stage
startDrag("ball", true, 0, 0, 225, 200);
```

See Also

`MovieClip._dropTarget`, *MovieClip.startDrag( )*, *stopDrag( )*

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Chapter 18. ActionScript Language Reference

stop( ) Global Function Flash 2

pause the movie's playhead at the current frame

```
stop()
```

Description

The *stop( )* function is a simple but fundamental function that halts the playback of a movie or movie clip. It is the global counterpart of the *MovieClip.stop( )* method and the arch-enemy of *play( )*. Commonly, it is used to wait for the user to choose from a graphical menu. Even while a movie is stopped, the Player continues to play sounds, react to user actions, and process events. Flash also continues to render changes to the screen in other movie clips or buttons, even if the playhead of the current movie clip is stopped. Similarly, visual changes performed in event handlers or via *setInterval( )* are rendered even when the playhead is stopped. The *stop( )* function merely prevents the playhead from advancing in the current timeline.

See Also

MovieClip.stop( ), play( )

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Chapter 18. ActionScript Language Reference

stopAllSounds( ) Global Function Flash 3

silence a movie

```
stopAllSounds()
```

Description

The *stopAllSounds( )* function mutes all the sounds currently playing in a movie, no matter how deeply the sounds are nested in movie clips. This applies to every sound in a movie, including programmatically generated *Sound* objects. For more precise control over stopping, starting, and setting the volume of sounds, see the *Sound* class.

Note that *stopAllSounds( )* has only a temporary effect. Any sound that starts after a *stopAllSounds( )* invocation will play normally. Because button sounds are not controllable via the *Sound* class, there is no way to mute an entire movie permanently. However, a Player-wide *Sound* object with a volume of 0 will silence all nonbutton sounds.

See Also

Sound.setVolume( ), *Sound.stop( )*

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

stopDrag( ) Global Function — Flash 4

end a drag operation in progress

```
stopDrag()
```

Description

The *startDrag( )* function causes a movie clip to follow the mouse pointer around the Stage. A *stopDrag( )* operation stops a dragged movie clip from following the mouse pointer. Because only one movie clip or movie can be dragged at a time, *stopDrag( )* does not require a `target` argument; it simply cancels any drag operation in progress.

Together with *startDrag( )*, *stopDrag( )* is used to create simple drag-and-drop interfaces in Flash, as demonstrated under "Interface Widgets" in the online Code Depot. If a dragged clip is dropped on top of another clip, the clip on which it was dropped is available through the `MovieClip._droptarget` property.

Example

The following code causes `window_mc` to be dragged while the mouse is pressed over it and dropped when the mouse is released. Note that we stop dragging the clip when either *onRelease( )* or *onReleaseOutside( )* is detected, ensuring that the clip is dropped, regardless of whether the mouse is actually over the clip when it is released:

```
window_mc.onPress = function () {
  this.startDrag();
}

window_mc.onRelease = window_mc.onReleaseOutside = function () {
  stopDrag();
}
```

See Also

`MovieClip._dropTarget`, *MovieClip.stopDrag( ), startDrag( )*

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

About Safari | Terms of Service | Privacy Policy | Contact Us | Help | Submit a Problem

Copyright © 2002 O'Reilly & Associates, Inc. All rights reserved.

1005 Gravenstein Highway North
Sebastopol, CA 95472

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Chapter 18. ActionScript Language Reference

#strict Pragma — Flash 5 (undocumented)

activates enhanced conformance to ECMA-26 standard

```
#strict
```

Description

An undocumented and volatile feature, the `#strict` pragma forces the Flash Player to conform more closely to the ECMA-262 specification, upon which ActionScript is based. It affects datatype conversion for the *undefined* and *string* datatypes, and it enforces complete case sensitivity.

> The `#strict` pragma is known to crash Internet Explorer on Windows under Flash Player 6 and should therefore not be used. At the time of this writing, Macromedia officially rescinded support for `#strict` in Flash Player 6.0.61.0 after the public beta test of Flash Player 6.0.60.48 revealed outstanding problems. For more details and the latest information on the status of `#strict`, see: http://www.moock.org/asdg/technotes/strict.

See Also

Appendix D

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Chapter 18. ActionScript Language Reference

String( ) Global Function — Flash 5

convert a value to the string datatype

```
String(value)
```

Arguments

value

An expression containing the value to be converted to a string.

Returns

The result of converting `value` to a primitive string.

Description

The *String( )* function converts its argument to a primitive string value and returns that converted value. The results of converting various types of data to a primitive string are described in Table 3-2 in Chapter 3. Normally it's not necessary to use the *String( )* function; ActionScript automatically converts values to the *string* type when appropriate.

Be sure not to confuse the *String( )* global function with the *String* class constructor. The former converts an expression to a string, whereas the later is a class that wraps primitive string data in an object so that properties and methods can be applied to it.

Usage

Note that the *String( )* function sometimes appears in Flash 4 *.fla* files that have been converted to a later Flash format. For information on how datatypes are handled when Flash 4 files are converted to Flash 5 or later, see Section C.3.4 in Appendix C.

See Also

The *String* class; "Explicit Type Conversion," in Chapter 3

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Chapter 18. ActionScript Language Reference

String Class Flash 6

wrapper class for string primitive typeFlash 5; performance enhanced in

Constructor

`new String(`*value*`)`

Arguments

value

An expression to be resolved and, if necessary, converted to a string, then wrapped in a *String* object.

Properties

length

The number of characters in a string.

Class Methods

The following method is invoked through the *String* class itself, not through an object of the *String* class:

fromCharCode( )

Generate a string from one or more Unicode code points.

Methods

The following object methods are invoked through an instance of the *String* class:

charAt( )

Retrieve a character at a specific position in the string.

charCodeAt( )

Retrieve the code point of a character at a specific position in the string.

concat( )

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Combine one or more items into a single string.

indexOf( )

Find the first occurrence of a specified substring in a string.

lastIndexOf( )

Find the last occurrence of a specified substring in a string.

slice( )

Extract a substring from a string, based on positive or negative character positions.

split( )

Convert a string to an array.

substr( )

Extract a substring from a string, based on a starting position and length.

substring( )

Extract a substring from a string, based on positive character positions only.

toLowerCase( )

Return a lowercase version of a string.

toUpperCase( )

Return an uppercase version of a string.

Description

The *String* class has several purposes:

- It allows us to access the `length` property of strings and perform string-related operations, such as *indexOf( )* and *slice( ). String* objects are created (and eventually deleted) automatically by the interpreter whenever a method is invoked on a primitive string value.
- It can be used to convert a datum of any type to a string.
- It is used to access the *fromCharCode( )* class method to create a new string, based on specified Unicode code points.
- It can be used to create a *String* object, which contains a primitive string value in an unnamed, internal property; however, there is little reason to do so.

In Flash 6, the performance (speed) of the *String* class methods is vastly improved over Flash 5 performance. Flash Player 6 executes most *String* methods 50 to 1000 times faster than Flash Player 5.

Usage

In practice, the *String* class constructor is used primarily to convert other datatypes to strings. See the *String( )* global function for more details.

See Also

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

"The String Type," in Chapter 4

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

String.charAt( ) Method Flash 5

retrieve the character from a specific position in a string

```
string.charAt(index)
```

Arguments

index

The integer position of the character to retrieve, which should be in the range of 0 (the first character) to `string .length - 1` (the last character).

Returns

The character in the position `index` within `string` .

Description

The *charAt ( )* method determines the character that resides at a certain position (`index`) in a string.

Example

```
trace("It is 10:34 pm".charAt(1));  // Displays: "t" (the second letter)
var country = "Canada";
trace(country.charAt(0));           // Displays: "C" (the first letter)

// This function removes all the spaces from a string and returns the result
function stripSpaces (inString) {
  var outString = "";
  for (i = 0; i < inString.length; i++) {
    if (inString.charAt(i) != " ") {
      outString += inString.charAt(i);
    }
  }
  return outString;
}
```

See Also

String.charCodeAt ( ) , *String.indexOf ( )* , *String.slice ( )* ; "The charAt( ) function," in Chapter 4

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Chapter 18. ActionScript Language Reference

String.charCodeAt( ) Method — Flash 5

retrieve the code point of the character at a specific position in a string

string.charCodeAt(*index*)

Arguments

index

The integer position of a character in *string* , which should be in the range of 0 (the first character) to *stri* .length - 1 (the last character).

Returns

An integer representing the Unicode code point of the character in the position *index* within *string* . For the code p the first 256 characters, see Appendix B.

Example

```
var msg = "A is the first letter of the Latin alphabet.";
trace(msg.charCodeAt(0));  // Displays: 65 (the code point for the "A" chara
trace(msg.charCodeAt(1));  // Displays: 32 (the code point for the space cha
```

See Also

String.charAt ( ) , *String.fromCharCode ( )* ; "The charCodeAt( ) function," in Chapter 4; Appendix B

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Chapter 18. ActionScript Language Reference

String.concat( ) Method — Flash 5

combine one or more values into a single string

```
string.concat(value1, value2,...valuen)
```

Arguments

value1 ,...value n

Values to be converted to strings (if necessary) and concatenated with `string` .

Returns

The result of concatenating `string` with `value1` , `value2` , ...`valuen` .

Description

The *concat ( )* method creates a string from a series of values. It is equivalent to using the concatenation operator (`+`) with strings but is sometimes preferred for clarity, as the `+` operator can also be used to add numbers. For details on *concat ( )* , see Chapter 4.

Usage

Note that *concat ( )* does not modify `string` ; it returns a completely new string.

Example

```
var greeting = "Hello";
excitedGreeting = greeting.concat("!");
trace(greeting);                          // Displays: "Hello"
trace(excitedGreeting);                   // Displays: "Hello!"

var x = 4;                                // Initialize x as an integer
trace(x + 5);                             // Displays: 9
trace(x.concat(5));                       // Fails because x is not a string
trace(String(x).concat(5));               // Displays: "45"

var x = "4";                              // Initialize x as a string
trace(x.concat(5));                       // Displays: "45"
trace(concat("foo", "fee"));              // Fails because concat( ) must be
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

```
                                                    // invoked as a method of a string
```

See Also

Section 4.6.1 and Section 4.6.1.1; Section 5.3.1

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Chapter 18. ActionScript Language Reference

String.fromCharCode( ) Class Method Flash 5

generate a string from one or more code points

```
String.fromCharCode(code_point1, code_point2,...code_pointn)
```

Arguments

code_ point1,...code_ pointn

A series of one or more integers corresponding to Unicode character code points.

Returns

A string formed by concatenating the characters represented by the specified code points.

Description

The *fromCharCode( )* class method produces a character or series of characters from character code points, as described in Chapter 4. For code points below 128, it effectively builds a string from a series of ASCII codes.

Example

```
// Makes a copyright symbol, followed by the year (2003)
copyNotice = String.fromCharCode(169) + " 2003";
```

See Also

String.charCodeAt( ); "The fromCharCode( ) function," in Chapter 4; Appendix B

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

String.indexOf( ) Method — Flash 5

find the first occurrence of a substring in a string

```
string.indexOf(substring)
string.indexOf(substring, startIndex)
```

Arguments

substring

A string containing the character or characters for which to search.

startIndex

An optional integer position in `string` at which to start searching for `substring`. Should be in the range of 0 (the first character) to `string.length - 1` (the last character). Defaults to 0.

Returns

The position of the first occurrence of `substring` in `string` (starting at `startIndex`). Returns -1 if `substring` is not found at or after `startIndex` in `string`.

Description

The *indexOf( )* method is used to search for characters in strings or to check whether a string contains a certain substring. The search is case-sensitive. Use *toLowerCase( )* to ensure a case-neutral search, as shown in the third Example.

Example

```
// Check if an email address contains an @ sign
var email = "derekaol.com";
if (email.indexOf("@") =  = -1) {
  trace("This isn't a valid email address");
}

// Check if an email address has an @ sign and is from the domain aol.com
var email = "derek@aol.com";
var atPos = email.indexOf("@");
if (atPos != -1 && email.indexOf("aol.com") =  = atPos + 1) {
  gotoAndStop("AOLuserOffer");
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

```
}
```

The following code creates a generic function that performs a case-insensitive search for a keyword in a string, as you might need when grading a fill-in-the-blank quiz:

```
// Generic function to search origStr for any occurrence
// of searchStr, using a case-insensitive comparison
function search (origStr, searchStr) {
  var origStr = origStr.toLowerCase();
  var searchStr = searchStr.toLowerCase();
  return origStr.indexOf(searchStr) != -1;
}

var answer = "einstein";
var guess = "Dr. Albert Einstein";

// Increment score if guess contains "einstein"
if (search(guess, answer)) {
  score++;
}
```

See Also

String.charAt( ), *String.lastIndexOf( )*, *String.toLowerCase( )*; "The indexOf( ) function," in Chapter 4

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

String.lastIndexOf( ) Method — Flash 5

find the last occurrence of a substring in a string

```
string.lastIndexOf(substring)
string.lastIndexOf(substring, startIndex)
```

Arguments

substring

A string containing the character or characters for which to search.

startIndex

An optional integer position in `string` at which to start the search for `substring`. The `string` is searched backward from `startIndex`, which should be in the range of 0 (the first character) to `string.length - 1` (the last character). Defaults to `string.length - 1`.

Returns

The position of the last occurrence of `substring` in `string` prior to `startIndex`. Returns -1 if `substring` is not found prior to `startIndex` in `string`.

Description

The *lastIndexOf( )* method is used to search for the last occurrence of a substring in a string or to check whether a string contains a certain substring. The search is case-sensitive. Use *toLowerCase( )* to ensure a case-neutral search, as shown in the Example under *String.indexOf( )*.

Example

```
URL = "http://www.moock.org/webdesign/flash/fillthewindow.html";
// Finds the last slash character
lastSlash = URL.lastIndexOf("/");
// Extracts the filename from the URL
file = URL.substring(lastSlash + 1);
trace(file);  // Displays: fillthewindow.html
```

See Also

String.charAt( ), *String.indexOf( )*, *String.toLowerCase( )*; "The lastIndexOf( ) Function," in Chapter 4

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Chapter 18. ActionScript Language Reference

String.length Property — Flash 5

the number of characters in a string — ***read-only***

string.length

Description

The `length` property returns the number of characters in *`string`* . Note that a null character (ASCII 0) does not signal the end of a string, as it would in some languages (such as C), but neither is it counted in the string's `length` . For example:

```
// Create the string "A" + null + "B"
var myString = String.fromCharCode(65,0,66);
trace(myString.length);  // Displays: 2 (The null character is ignored)
```

Example

```
var myString = "hello";
trace (myString.length);  // Displays: 5
trace ("hello".length);   // Displays: 5

// Here we convert the number 1000 to a string in order to test its length
var age = 1000;
// Display an error message if the number has the wrong number of digits
if (String(age).length != 2) {
  trace ("Please enter a two-digit number");
}
```

See Also

Array.length ( ) ; "The length property," in Chapter 4

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

String.slice( ) Method — Flash 5

extract a substring from a string, based on positive or negative character positions

```
string.slice(startIndex, endIndex)
```

Arguments

startIndex

The integer position of the first character to extract from `string`. If `startIndex` is negative, the position is measured from the end of the string, where -1 is the last character, -2 is the second-to-last character, and so on (i.e., a negative `startIndex` specifies the character at `string.length + startIndex`).

endIndex

The integer position of the character *after* the last character to extract from `string`. If `endIndex` is negative, the position is measured from the end of the string, where -1 is the last character, -2 is the second-to-last character, and so on (i.e., a negative `endIndex` specifies the character at `string.length + endIndex`). Defaults to `string.length` if omitted.

Returns

A substring of `string`, starting at `startIndex` and ending at `endIndex - 1`, where both `startIndex` and `endIndex` are zero-relative.

Description

The *slice( )* method is one of three methods that can be used to extract a substring from a string (the others being *substring( )* and *substr( )*). The *slice( )* method offers the option of using negative start and end index values, which allows us to extract a substring by measuring back from the end of a string.

Usage

Note that *slice( )* does not modify `string`; it returns a completely new string.

Example

```
var fullName = "Steven Sid Mumby";
middleName = fullName.slice(7, 10);   // Assigns "Sid" to middleName
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

```
middleName = fullName.slice(-9, -6);  // Also assigns "Sid" to middleName
```

See Also

String.substr( ), *String.substring( )*; "The slice( ) function," and "Combining String Examination with Substring Extraction," in Chapter 4

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

String.split( ) Method Flash 5

convert a string into a series of array elements

```
string.split(delimiter)
```

Arguments

delimiter

The character or series of characters at which to break `string` when forming elements of the new array.

Returns

An array whose elements contain the substrings formed by breaking `string` into segments demarcated by `delimite`

Description

The *split ( )* method breaks a string into substrings, assigns those substrings to the elements of an array, and returns that ar Contiguous occurrences of `delimiter` without intervening characters result in empty elements. For example, the follc code:

```
owners = "terry,doug,,,jon";
ownersArray = owners.split(",");
```

assigns the following elements to `ownersArray` (elements 2 and 3 contain `undefined`):

```
0: terry
1: doug
2:
3:
4: jon
```

The *split ( )* method typically is used to convert a string received from a CGI script or text file into an array for further manipulation. It is also useful when parsing the parameters of an *asfunction* call from an HTML text field `<A>` tag, which pass only one string argument to a function. See Section E.7 in Appendix E. Commonly used delimiters include the comm the tab character. Use a space as the delimiter to split a string into a series of words.

Example

Suppose we store a list of names in a text file named *names.txt* . Each name is separated from the others by a Tab characte implied by the whitespace shown:

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

```
owners=terry    doug    jon
```

On frame 1 of our movie, we load the *names.txt* file into our movie:

```
this.loadVariables("names.txt");
```

After ensuring that *names.txt* has fully loaded (see *loadVariables( )*), we split the loaded `owners` variable into an array:

```
splitString = String.fromCharCode(9);  // Assign the Tab character to splitS
ownersArray = owners.split(splitString);

trace(ownersArray[1]);  // Displays: "doug"
```

Note that in Flash 5 *split( )* can take a long time to execute with large bodies of text; it is considerably faster in Flash 6. If performance is a problem, break your data into manageable portions or consider using XML instead. See the *XML* class.

Bugs

In Flash 5, using the empty string as a delimiter adds all of *`string`* to the first element of the array being generated. According to ECMA-262, an empty string delimiter should cause *`string`* to be broken at each character. Similarly, multicharacter delimiters are not recognized by Flash 5 and cause all of *`string`* to be assigned to the first element of t returned array. These bugs are fixed in Flash Player 6.

See Also

Array.join( ) ; "The split( ) function," in Chapter 4

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

String.substr( ) Method — Flash 5

extract a substring from a string, based on a starting position and length

```
string.substr(startIndex, length)
```

Arguments

startIndex

The integer position of the first character to extract from `string` . If `startIndex` is negative, the position is measured from the end of the string, where -1 is the last character, -2 is the second-to-last character, and so on (i.e., a negative `startIndex` specifies the character at `string .length + startIndex`).

length

The number of characters to extract from `string` , starting at (and including) the `startIndex` . If not specified, all the characters from `startIndex` to the end of `string` are extracted.

Returns

A substring of `string` , starting at `startIndex` and including `length` characters. If `length` is omitted, the result contains characters from `startIndex` to the end of `string` .

Description

The *substr ( )* method is one of three methods that can be used to extract a substring from a string (the others being *slice( )* and *substring( )*). The *substr( )* method extracts a string based on the number of characters specified by `length` , not based on an ending character index.

Usage

Note that *substr ( )* does not modify `string` ; it returns a completely new string.

Example

```
var fullName = "Steven Sid Mumby";

middleName = fullName.substr(7, 3);       // Assigns "Sid" to middleName
firstName  = fullName.substr(0, 6);       // Assigns "Steven" to firstName
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

```
lastName   = fullName.substr(11);          // Assigns "Mumby" to lastName

// Notice the negative starting indexes...
middleName = fullName.substr(-9, 3);       // Assigns "Sid" to middleName
firstName  = fullName.substr(-16, 6);      // Assigns "Steven" to firstName
lastName   = fullName.substr(-5);          // Assigns "Mumby" to lastName
```

See Also

String.slice ( ) , *String.substring ( )* ; "Combining String Examination with Substring Extraction," and "The substr( ) function," in Chapter 4

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Chapter 18. ActionScript Language Reference

String.substring( ) Method — Flash 5

extract a substring from a string, based on positive character positions

string.substring(*startIndex*, *endIndex*)

Arguments

startIndex

The positive integer position of the first character to extract from `string` . If negative, 0 is used.

endIndex

The positive integer position of the character *after* the last character to extract from `string` . Defaults to `stri` `.length` if omitted. If negative, 0 is used.

Returns

A substring of `string` , starting at `startIndex` and ending at `endIndex - 1` , where both `startIndex` a `endIndex` are zero-relative.

Description

The *substring ( )* method is one of three methods that can be used to extract a substring from a string (the others being *slic* and *substr( )*). The *substring( )* method is identical to *slice ( )* , except that it does not allow for negative `startIndex` `endIndex` values and it automatically reorders the two indexes if `endIndex` is less than `startIndex` .

Usage

Note that *substring ( )* does not modify `string` ; it returns a completely new string.

Example

```
// Extract names from a string
var fullName = "Steven Sid Mumby";
middleName = fullName.substring(7, 10);   // Assigns "Sid" to middleName
middleName = fullName.substring(10, 7);   // Assigns "Sid" to middleName
                                          // (indexes are swapped automatica
firstName  = fullName.substring(0, 6);    // Assigns "Steven" to firstName
lastName   = fullName.substring(11);      // Assigns "Mumby" to lastName
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Example 18-4 is a reusable function to search for and replace all occurrences of a substring within a string.

Example 18-4. A search-and-replace function

```
function replace (origStr, searchStr, replaceStr) {
  var tempStr = "";
  var startIndex = 0;
  if (searchStr =  = "") {
    return origStr;
  }

  if (origStr.indexOf(searchStr) != -1) {
    while ((searchIndex = origStr.indexOf(searchStr, startIndex)) != -1) {
      tempStr += origStr.substring(startIndex, searchIndex);
      tempStr += replaceStr;
      startIndex = searchIndex + searchStr.length;
    }
    return tempStr + origStr.substring(startIndex);
  } else {
    return origStr;
  }
}

msg = "three times three is four";
trace(replace(msg, "three", "two"));  // Displays: "two times two is four"
```

See Also

String.slice ( ) , *String.substr ( )* ; "Combining String Examination with Substring Extraction," and "The substr( ) function in Chapter 4

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

String.toLowerCase( ) Method Flash 5

generate a lowercase version of a string

```
string.toLowerCase()
```

Returns

The lowercase equivalent of *string* as a new string. Characters without a lowercase equivalent are left unchanged.

Description

The *toLowerCase( )* method creates a new, lowercase version of *string*; it can be used for formatting or for facilitating case-insensitive character comparisons.

Usage

Note that *toLowerCase( )* does not modify *string*; it returns a completely new string.

Example

```
// Set msg to "this sentence has mixed caps!"
msg = "ThiS SenTencE Has MixED CaPs!".toLowerCase();

// Perform a case-insensitive comparison of two strings
function caseInsensitiveCompare (stringA, stringB) {
  return (stringA.toLowerCase() =  = stringB.toLowerCase());
}

trace(caseInsensitiveCompare("Colin", "colin"));  // Displays: true
```

See Also

String.toUpperCase( ); "The toLowerCase( ) function," in Chapter 4

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Chapter 18. ActionScript Language Reference

String.toUpperCase( ) Method Flash 5

generate an uppercase version of a string

string.toUpperCase( )

Returns

The uppercase (a.k.a. ALL CAPS) equivalent of *string* as a new string. Characters without an uppercase equivalent are left unchanged.

Description

The *toUpperCase( )* method creates a new, uppercase version of *string*; it can be used for formatting or for facilitating case-insensitive character comparisons.

Usage

Note that *toUpperCase( )* does not modify *string*; it returns a completely new string.

Example

```
"listen to me".toUpperCase();   // Yields the string "LISTEN TO ME"
var msg1 = "Your Final Score: 234";
var msg2 = msg1.toUpperCase();  // Set msg2 to "YOUR FINAL SCORE: 234"
```

See Also

String.toLowerCase( ); "The toUpperCase( ) function," in Chapter 4

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Chapter 18. ActionScript Language Reference

super "Operator" Flash 6

invoke an object's superclass constructor, or invoke a superclass's implementation of an overridden method

```
super(arg1, arg2,...argn);
super.methodName(arg1, arg2,...argn);
```

Operands

arg1, ...argn

A list of arguments passed to the superclass constructor function.

methodName

The superclass's implementation of an overridden method to be invoked.

Description

The *super* "operator," added in Flash 6, serves two distinct object-oriented programming purposes:

- It can invoke an object's superclass constructor.
- It can invoke a superclass's implementation of an overridden method.

See "The super `Operator'" in Chapter 5 for many more details.

Usage

Technically, *super* is not a real operator in ActionScript, which is why it is presented here as the *super* "operator."

See Also

instanceof, *new*, *Object.hasOwnProperty( )*; "The super `Operator'," in Chapter 5; "Making a superclass," "Accessing overridden methods," and "A custom setSuperClass( ) function,"in Chapter 12.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Chapter 18. ActionScript Language Reference

System Object — Flash 6

access to Player and system settings and specifications

Properties

capabilities

A reference to the *Capabilities* object.

security

Object for setting cross-domain movie permissions.

useCodepage

Force Flash Player to use the operating system's code page rather than Unicode.

Methods

showSettings( )

Display the Flash Player Settings dialog box.

Description

The *System* object provides information about, and control over, features of the Flash Player and the system on which it is running. With the exception of *showSettings ( )* (used to display the Flash Player Settings dialog box) and useCodepag (used to control the character encoding mode), Player and system features are accessed through the *Capabilities* and *Secur* objects, which themselves are stored as properties of *System* . The *Capabilities* object is documented separately as the *Capabilities Object* (under the letter "C") in this *Language Reference* . The *Security* object's single method, *allowDomain* which can permit movies loaded from different domains to access and control each other, is documented under *System.security.allowDomain( )* .

Note that *System* is a predefined object, not a class. Its properties and methods are accessed directly, as in:

```
System.showSettings(0);  // Displays the Privacy tab of the Settings dialog
```

See Also

The *Capabilities* object, the *Stage* object

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Chapter 18. ActionScript Language Reference

System.capabilities Property Flash 6

a reference to the Capabilities object ***read/write***

```
System.capabilities
```

Description

The `capabilities` property is an object reference to the built-in *Capabilities* object. See the *Capabilities* object entry (under "C") in this *Language Reference* for complete details. Although, technically, `System.capabilities` is writable, it should not be overwritten in your movies.

See Also

The *Capabilities* object

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Chapter 18. ActionScript Language Reference

System.security.allowDomain( ) Method Flash 6

sets cross-domain movie permissions

```
System.security.allowDomain(domain1, domain2,...domainn)
```

Arguments

domain1 , ...domain n

A list of one or more domains for which access to the movie should be granted.

Description

The `security` property is a reference to the built-in *Security* object, which defines a single important method, *allowDo* We use *allowDomain( )* to grant movies at other domains access to the current movie. Though security limitations have be part of the Flash Player since version 4, Flash 6 formalizes the concept of the *security sandbox* . Under the rules of the sa movie hosted at one domain cannot:

- Invoke methods or functions in a movie loaded from another domain
- Access data (user-defined variables, user-defined properties, and nested movie clips) in a movie loaded from anothe
- Load XML or variables from another domain via the *LoadVars* class, *loadVariables( )* , *XML.load( )* , or *XML.sendAndLoad( )*
- Connect to an *XMLSocket* server on another domain

By default, the security sandbox is active, and all attempts to perform the above operations are ignored by Flash Player 6. *allowDomain ( )* , we can override these restrictions.

The *allowDomain ( )* method is invoked by the movie that wishes to grant access to a particular domain or domains, speci `domain1` , `domain2` , ...`domainn` . For example, suppose Graham posts his resume, *grayResume.swf* , on *graydie* and wants to allow *jobBoard.swf* , posted on *moock.org* , to access it. On the first frame of *grayResume.swf* , he must use *allowDomain ( )* to grant movies at *moock.org* access, as follows:

```
System.security.allowDomain("moock.org");
```

Once *moock.org* has been added to *grayResume.swf* 's list of trusted domains, *jobBoard.swf* (at *moock.org*) can load *grayResume.swf* from *graydient.com* and set its variables, stop its playhead, or otherwise manipulate it without any securi limitations. For example:

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

```
// Make a container into which to load grayResume.swf
this.createEmptyMovieClip("holder", 1);
// Load grayResume.swf
holder.loadMovie("http://www.graydient.com/grayResume.swf");

// Create a preloader that detects when the movie has loaded
var preloadInterval = setInterval(setText, 1000);

// Function to be called every second. When the movie has loaded,
// the function assigns it some text and stops its playhead.
function setText () {
  if (holder._framesloaded > 0 && holder._framesloaded =  = holder._totalfra
    // Stop the loaded movie
    holder.stop();
    // Assign some text
    holder.subtitle_txt.text = "Gray's resume brought to you by moock.org.";
    // Stop the load checker
    clearInterval(preloadInterval);
  }
}
```

Movies that are hosted at the same domain are not affected by sandbox restrictions. Table 18-21 demonstrates when the se sandbox comes into effect. To check the domain of the current movie, use *LocalConnection.domain( )* .

Table 18-21. Domain-based security restrictions

Domain of movie origin	Host to connect to	Permitted?	Permitted via *allowDomain( )* ?
www.somewhere.com	*www.somewhere.com*	Yes	Yes
www.somewhere.com	*other.somewhere.com*	Yes	Yes
www.somewhere.com	*somewhere.com*	Yes	Yes
somewhere.com	*www.somewhere.com*	Yes	Yes
www.somewhere.com	*www.elsewhere.com*	No	Yes

For backward compatibility, Flash 4 and Flash 5 movies loaded from different domains into Flash Player 6 can still access other's data (as was possible in the older versions of the Player). However, a Flash 5 movie cannot access the data in a Fla movie loaded from a different domain, nor vice versa.

The *allowDomain ( )* method cannot directly override security restrictions on loading XML, loading variables, or commur over *XMLSocket* . However, it is possible to load XML or variables from a different domain by using a movie on that dor proxy for communications. For example, suppose *main.swf* at *moock.org* wants to load a file, *newsFeed.xml* , from *graydi* The *main.swf* file must load a proxy movie, *xmlloader.swf* , from *graydient.com* . Then, *xmlloader.swf* must load the *newsFeed.xml* file and use *allowDomain ( )* to grant *moock.org* access to it. The *main.swf* movie can then access the load via *xmlloader.swf* . Unfortunately, at the time of writing, a bug prevents this proxy-movie system from working in the cur build of Flash Player 6 (6.0.61.0). Macromedia is aware of the issue and is working to address it.

XML and loaded variable security can also be circumvented with a either a DNS alias or a server-side proxy script that ac between for Flash and the external site. For more information, see:

http://www.macromedia.com/support/flash/ts/documents/loadvars_security.htm
http://www.macromedia.com/support/flash/ts/documents/load_xdomain.htm

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

A Flash client at one domain can connect to an *XMLSocket* server on a different domain by using a movie on the server's as a proxy for communications (the bug mentioned earlier does not affect *XMLSocket* communications). For example, suj chat socket server is running on *moock.org* . The owner of *graydient.com* wants to post a chat movie (*chat.swf*) at *graydie* that uses *moock.org* 's chat server.

The owner of *moock.org* must create a proxy movie for *graydient.com* by following these steps:

1. Create a movie called *socketObject.swf* .
2. On frame 1 of *socketObject.swf* , add the following code:

```
// Grant access to graydient.com
System.security.allowDomain("graydient.com");
// Create the XMLSocket object. This must be done here.
sock = new XMLSocket();
```

3. Post *socketObject.swf* at moock.org (the socket server's domain).

From *chat.swf* , the owner of *graydient.com* must use the *socketObject.swf* proxy movie to communicate with *moock.org* server, as follows:

1. On frame 1 of *chat.swf* , load *socketObject.swf* onto level 1 (any level or target will work):

```
loadMovieNum("http://www.moock.org/socketObject.swf", 1);
```

2. Once *socketObject.swf* has loaded, use the `_level1.sock` object to connect to and communicate with the sock For example:

```
// Create an output text field
this.createTextField("out_txt", 1, 100, 100, 400, 100);
out_txt.border = true;

// Set callback to handle connection
_level1.sock.onConnect = function (success) {
  out_txt.text += ("\n Connected: " + success);
}
// Set callback to handle incoming XML
_level1.sock.onXML = function (doc) {
  out_txt.text += ("\n XML arrived: " + doc);
}
// Make the connection
_level1.sock.connect("moock.org", 2001);

// Code for a button that sends XML to the socket server
sendMessage_btn.onPress = function () {
  _level1.sock.send("<MESSAGE>hello world</MESSAGE>");
}
```

A movie running off the local filesystem in the Flash Standalone Player or browser-based Player (i.e., plugin) is not subje security sandbox. It can load XML or variables from any domain, and it can control and access the data of any movie it lo However, the opposite is not true. For example, if a local movie loads another movie, *booklist.swf* , from oreilly.com, *boo* cannot access or control the local movie unless the local movie includes the following call to *allowDomain ( )* :

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

```
System.security.allowDomain("oreilly.com");
```

The general rule of thumb is, "local can control loaded, but loaded cannot control local."

Note that built-in properties, such as `_framesloaded` and `_x` , are not restricted by the sandbox. For example, a movi loads another movie can change the position of the loaded movie regardless of its domain of origin.

See Also

loadMovie( ) , *loadVariables( )* , the *LoadVars* class, *XML.load ( )* , *XML.sendAndLoad ( )* , the *XMLSocket* class

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Chapter 18. ActionScript Language Reference

System.showSettings( ) Method Flash 6 (undocumented)

displays the Flash Player Settings dialog box

```
System.showSettings(tabID)
```

Arguments

tabID

An integer indicating the specific tab to display.

Description

The *showSettings( )* method displays the Macromedia Flash Player Settings dialog box, from which the user can set per-movie Player options. In Flash Player 6, the Settings dialog box includes four tabs, each of which corresponds to an integer `tabID`, as follows: Privacy (0), Local Storage (1), Microphone (2), and Camera (3).

Normally, the user must access the Macromedia Flash Player Settings dialog box manually by right-clicking on the movie under Windows (Ctrl-click on Macintosh) and selecting "Settings . . . " from the contextual menu. The *showSettings( )* method lets us assist the user by displaying the settings dialog box programmatically when asking the user to change an option required by a movie. For example, if a movie needs more than 100 KB of local disk space to run, we can ask the user to increase the limit, and then we can issue the following command to open the Settings dialog box to the tab where the user can specify the local disk storage limit:

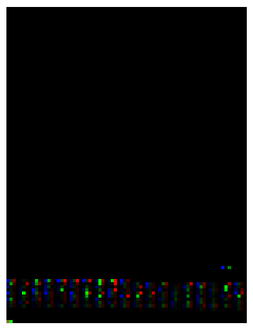
```
System.showSettings(1);  // Display Local Storage settings tab
```

See Also

SharedObject.flush( ), SharedObject.getSize( ), `Stage.showMenu`

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir



Chapter 18. ActionScript Language Reference

System.useCodepage Property — Flash 6

force Flash Player to use the operating system's code page rather than Unicode — *read/write*

`System.useCodepage`

Description

Unicode is an international character set comprising all the characters of the world's languages. Most modern applications, including Flash Player 6, use Unicode to manage and display text. Older applications, including Flash Player 5, manage text using the operating system's so called *code page* (essentially, a regional character set).

The `System.useCodepage` property controls:

- Display of text fields
- *XML.load( )*, *XML.send( )*, and *XML.sendAndLoad( )* operations
- *LoadVars.load( )*, *LoadVars.send( )*, and *LoadVars.sendAndLoad( )* operations
- *getURL( )*, *loadMovie( )*, and *loadVariables( )* actions that transmit/receive variables
- *XMLSocket* communications
- *escape( )* and *unescape( )* functions

When a non-Unicode application renders a character to the screen, it looks up the character by its code point in the system's current code page. The character displayed on screen is the one specified by the code page in use at display time. If the input code page (the one in effect when the character was input) does not match the display code page (the one in use when the application renders the character), the input character specified will likely not match the character displayed. The Unicode standard was introduced to solve this problem. Because Flash 6 uses Unicode, you should have no need for the `System.useCodepage` property as long as you:

- Use Unicode (encoded using either UTF-8 or UTF-16) for all external text
- Use a Unicode escape sequence to include any character you cannot enter via your keyboard in the Flash authoring tool

The Boolean `useCodepage` property can force Flash Player 6 to use the operating system's current code page, rather than Unicode, when rendering text, as follows:

downloaded from: lib.ommolketab.ir

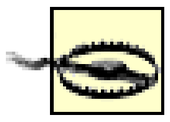
downloaded from: lib.ommolketab.ir

```
// Force use of the operating system's code page
System.useCodepage = true;
```

`System.useCodepage` can be modified at any time—for example, at the beginning and end of an operation that requires a legacy text encoding, such as loading an existing text file that uses a specific encoding.

When exporting Flash 6-format *.swf* files from the Flash MX authoring tool, it rarely is necessary to use the `useCodepage` property (Flash MX automatically converts text entered in the authoring tool to Unicode upon export). However, when the following conditions are both true, setting the `useCodepage` property to `true` will force non-Unicode text to display as intended:

- Non-Unicode text is loaded into the Flash Player, from either an external text file or a server script.
- The external text happens to be encoded with the same code page as is currently set on the operating system running the Flash Player.



The text will not display correctly if the operating system's current code page does not match the input code page! Though forcing the use of the operating system's code page may appear to magically fix text display problems on one computer, setting `System.useCodepage` to `true` almost never suitably addresses international text display issues. Hence, whenever possible, all external files should use Unicode, and `useCodepage` should be left at its default setting, `false`.

By default, most western Windows computers use code page CP1252 (WinLatin1). Mappings from CP1252 to Unicode can be found at:

http://www.unicode.org/Public/MAPPINGS/VENDORS/MICSFT/WINDOWS/CP1252.TXT

By default, most Japanese Windows computers use code page CP932 (Windows Shift-JIS). Mappings from CP932 to Unicode can be found at:

http://www.unicode.org/Public/MAPPINGS/VENDORS/MICSFT/WINDOWS/CP932.TXT

The complete list of Windows code pages is available at:

http://www.microsoft.com/globaldev/reference/WinCP.asp

For a partial list of code pages used by other vendors see:

http://czyborra.com/charsets/codepages.html

For more information on Unicode in Flash, see Section 4.5.1 in Chapter 4.

See Also

Section 4.5

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Chapter 18. ActionScript Language Reference

targetPath( ) Global Function Flash 5

the absolute path to a movie or movie clip

```
targetPath (movieClip)
```

Arguments

movieClip

A reference to a movie clip object.

Returns

A string representing the path to `movieClip` in absolute terms, using dot notation (e.g., `"_level0.myMovie"`).

Description

The *targetPath( )* function returns a movie clip's reference as a string that describes the absolute path to the clip (identical to the return value of *MovieClip.valueOf( )*). The *targetPath( )* function sometimes is used to compose placement-sensitive code that operates on a movie clip, relative to the timeline upon which the clip resides. To retrieve the slash-notation path to a movie clip (rather than *targetPath( )*'s dot notation), use `MovieClip._target`.

Example

If `square` is a movie clip contained by `shapes`, which resides on the main timeline of `_level0`, then inside the `shapes` clip, the statement:

```
targetPath(square);
```

returns:

```
"_level0.shapes.square"
```

See Also

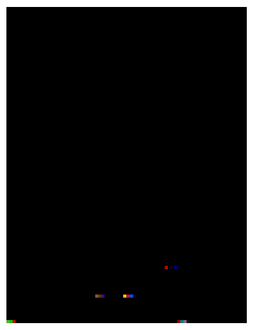
MovieClip._target, *MovieClip.valueOf( )*; "The targetPath( ) function," in Chapter 13

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir



Chapter 18. ActionScript Language Reference

tellTarget( ) Global Function — Flash 5

execute statements in the scope of a remote movie clip"Tell Target tool" in Flash 3; deprecated in

```
tellTarget (target) {
  statements}
```

Arguments

target

A string or reference indicating the path to a movie or movie clip instance (references are converted to paths when used in a string context).

statements

The statements to be executed in the scope of `target`.

Description

In Flash 3 and Flash 4, *tellTarget( )* was the primary means of communicating between two movie clips (i.e., controlling one from the other). It was used to invoke functions such as *play( )*, *stop( )*, and *gotoAndStop( )* on remote movie clips. In Flash 4, when variables were added to ActionScript, we could also use *tellTarget( )* to get and set remote clip variable values. As of Flash 5, these endeavors are better accomplished with the dot operator, ., and the array access operator, `[ ]`. Another alternative to the *tellTarget( )* function is the *with* statement, described in Chapter 6.

Usage

The *tellTarget( )* function may be better described as a statement because it requires a substatement block. However, the point is academic, as *tellTarget( )* has been deprecated.

Example

```
// Move the ball movie clip over 300 pixels to the right and
// send its playhead to the frame labeled "redStripes".
tellTarget ("ball") {
  gotoAndStop("redStripes");
  _x += 300;
}
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

See Also

"The with Statement," in Chapter 6; "The `Objectness' of Movie Clips," in Chapter 13; "Whither Tell Target?," in Appendix C

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Chapter 18. ActionScript Language Reference

TextField Class Flash 6

display and manipulate text on screen

Constructor

None. *TextField* objects are created manually in the authoring tool or programmatically via the *MovieClip.createTextField( )* method.

Properties

_alpha

Opacity percentage: 0 is transparent; 100 is opaque.

autoSize

Matches the text field's bounding rectangle size to its text.

background

Boolean; turns text field background on and off.

backgroundColor

RGB number specifying the text field's background color.

border

Boolean; turns text field border on and off.

borderColor

RGB number specifying the text field's border color.

bottomScroll

The 1-relative index of the text field's lowest visible line.

condenseWhite

Boolean; enables or disables condensing of whitespace in HTML text.

embedFonts

Boolean; renders text using fonts exported with the movie.

_height

Height of the text field's bounding box, in pixels.

hscroll

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Text's horizontal scroll position, in pixels.

html

Boolean; enables or disables HTML display.

htmlText

HTML source code to render in the text field.

length

The number of characters in the text field.

maxChars

Limits the allowed length of user input.

maxhscroll

The farthest position text can be scrolled to the left.

maxscroll

The last legal top line of a text field.

multiline

Boolean; enables or disables multiple-line user input.

_name

The instance name of the *TextField* object.

_parent

A reference to the movie clip in which the text field resides.

password

Boolean; obscures characters displayed in the field.

restrict

Limits text entry to a series of characters.

_rotation

Rotation, in degrees, of the text field.

scroll

The current top line displayed in the text field.

selectable

Boolean; enables or disables selection by the user.

tabEnabled

Boolean; includes or excludes the text field from the current tab order.

tabIndex

Specifies the text field's index in the custom tab order.

_target

The target path of the text field, in slash syntax.

text

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Specifies the characters to be rendered in the field.

textColor

Sets the color for all characters in the text field.

textHeight

Pixel height of all the text in the text field.

textWidth

Pixel width of the longest line in the text field.

type

Specifies whether the field can accept user input.

_url

The network address of the movie clip that contains the field.

variable

Associates a variable with the text property.

_visible

Boolean; whether the text field is shown or hidden.

_width

Width of the text field's bounding box, in pixels.

wordWrap

Boolean; enables or disables automatic line-breaking for long lines.

_x

Horizontal location of the text field, in pixels.

_xmouse

Horizontal location of mouse pointer, relative to the text field.

_xscale

Width of the text field, as a percentage of original.

_y

Vertical location of the text field, in pixels.

_ymouse

Vertical location of mouse pointer, relative to the text field.

_yscale

Height of the text field, as a percentage of original.

Class Methods

getFontList( )

Returns an array of fonts on the user's system.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Methods

addListener( )

Registers an object to receive *onChanged( )* and *onScroller( )* events.

getDepth( )

Returns the position of the text field in the movie stack.

getNewTextFormat( )

Retrieves the default *TextFormat* object for the text field.

getTextFormat( )

Retrieves a *TextFormat* object for specified characters.

removeListener( )

Cancels event notices for the specified listener.

removeTextField( )

Deletes a runtime-created text field.

replaceSel( )

Replaces selected text (without disturbing formatting).

setNewTextFormat( )

Sets the default formatting for the text field.

setTextFormat( )

Sets the formatting of specified characters.

Event Handlers

onChanged( )

Callback invoked when user input is detected.

onKillFocus( )

Callback invoked when the field loses focus.

onScroller( )

Callback invoked when any scroll property changes.

onSetFocus( )

Callback invoked when the field gains focus.

Listener Events

onChanged( )

Occurs when user input is detected.

onScroller( )

Occurs when any scroll property changes.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Description

The *TextField* class provides control over text displayed on screen. Each *TextField* object represents a rectangular text container that can be filled with text, sized, or formatted, and can even receive user input. When we speak of a "text field," we mean the rectangular container on screen; when we speak of a *TextField* object or instance, we mean the ActionScript object that controls a text field.

TextField objects are not constructed with the *new* operator; instead, they are created automatically by Flash whenever author-time text is drawn (with the Text tool) or when the *MovieClip.createTextField( )* method is invoked. For example, the following code creates a text field named `theField_txt` , and places it on depth 1, at Stage coordinates (0,0), with a width of 200 and a height of 20. The `this` keyword refers to the current movie clip. The suffix "_txt" activates code hinting in the Actions panel:

```
this.createTextField("theField_txt", 1, 0, 0, 200, 20);
```

In order to gain access to a *TextField* instance for a field created at authoring time, we must set its type to Dynamic Text or Input Text and assign it an instance name via the Property inspector.

The following steps describe the typical workflow for creating an ActionScript-controllable text field at authoring time:

1. Select the Text tool.
2. Drag a rectangle big enough to display the desired text on stage .
3. Make sure the Property inspector (Window Properties) is open.
4. In the Property inspector, for Type, select Dynamic Text or Input Text (the default type, Static Text, cannot be accessed through ActionScript).
5. In the Property inspector, for Instance Name, type **`someName_txt`** . An Instance Name cannot be specified until the Type is set, as described in Step 4.

For instructions on deleting a text field at runtime, see the *TextField .removeTextField( )* method.

Plain text displayed in a text field is retrieved through the `text` property and set through either the *replaceSel( )* method or the `text` property. HTML text can be set through the `htmlText` property, provided that the `html` property is also set.

> Prior to Flash 6, a text field's ActionScript identifier (known as its "variable") was synonymous with the content of the field, but in Flash 6, a text field's identifier is an object instance name. The contents of the text field are retrieved and set in Flash 6 via the *`TextField`* `.text` property.

Hence, in Flash 5, we used the following syntax to set the `highScore_txt` text field to "15, 000":

```
highScore_txt = "15, 000";
```

But in Flash 6, we must use the `text` property to set a text field's text:

```
highScore_txt.text = "15, 000";
```

The Var: text box in Flash MX's text field Property inspector is retained for backward compatibility, as is the *`TextField`* `.variable` property. When authoring for Flash Player 6, assign all text fields an instance name and use the `text` property to access their content.

By default, Dynamic and Input text fields are selectable by the user. To create a nonselectable text field for display only,

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

set its `selectable` property to `false` :

```
output_txt.selectable = false;
```

When a user selects a portion of a dynamic or user-input text field, the indexes of the selected characters are stored in the *Selection* object. Using the *Selection* object, we can check which part of a text field a user has selected; we can even select a part of a text field programmatically. The *Selection* object can also give keyboard focus to a particular text field or tell us which text field is currently selected by the user. See *Selection.onSetFocus( )* , *TextField.onKillFocus( )* , and *TextField.onSetFocus( )* .

For author-time text fields, many *TextField* object properties can be set through the Property inspector interface; any options set at authoring time are reflected at runtime by the corresponding *TextField* instance's properties.

We can respond to changes in a text field's scroll position or content via the *TextField* events *onChanged( )* and *onScroller( )* .

To format the appearance of text in a text field, we use either a *TextFormat* object or HTML display. For details, see the following *TextField* entries: `embedFonts` , *getFontList( )* , *getTextFormat( )* , *getNewTextFormat( )* , `html` , `htmlText` , *setFontList( )* , *setTextFormat( )* , *setNewTextFormat( )* , and *textColor( )* , and see the *TextFormat* class entry. The border and background color of a text field's containing rectangle can also be set; see the following *TextField* entries: `border` , `borderColor` , `background` , and `backgroundColor` .

Prior to Flash 6, text fields were not formally accessible as objects. As of Flash 6, the *TextField* class lets us manipulate text fields much as we do movie clips, by setting position, size, rotation, etc. In fact, many text field properties are borrowed directly from the *MovieClip* class (for example, `_x` , `_y` , `_width` , `_height` , and `_rotation`). For the sake of backward compatibility, text field properties corresponding to movie clip properties that existed prior to Flash 6 are prefixed with an underscore (_). To conform more closely with other ECMA-262 languages, properties that are completely new in Flash 6 do not use the underscore convention.

Bugs

In Flash 6, built-in properties set on `TextField.prototype` are not inherited by text field instances, so custom default values cannot be assigned for all text fields in a movie:

```
// This attempt to give all text fields a border by default fails in Flash 6
TextField.prototype.border = true;
```

See Also

MovieClip.createTextField( ) , the *TextFormat* class; Chapter 17

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Chapter 18. ActionScript Language Reference

TextField.addListener( ) Method Flash 6

registers an object to receive onChanged( ) and onScroller( ) events

```
theField.addListener(listener)
```

Arguments

listener

An object with an *onChanged( )* and/or *onScroller( )* method defined.

Returns

A Boolean indicating whether the listener was added successfully. In practice, *addListener( )* always returns `true` in Flas Player 6. However, in future versions it may return `false` when not enough memory is available to add the listener.

Description

The *addListener( )* method adds `listener` to the list of objects notified when `theField` 's *onScroller( )* or *onChan* events fire. The `listener` argument is an object that defines methods of the name *onScroller* and *onChanged* ; these a invoked when the corresponding events occur for `theField` . To stop a listener object from receiving events, use *TextField.removeListener( )* .

Listener events allow a single *TextField* to trigger functions on multiple objects. When only one function needs to be invo in response to a *TextField* event, a callback handler (which offers simpler syntax) will suffice. Note that a *TextField* can h callbacks and listeners for the same event. Callback functions are executed before listener methods. When multiple listene registered to a *TextField* , their methods are invoked in the order the listeners were added.

For general information on how listeners and callback functions work, see Chapter 10.

Example

The following code creates a *TextField* , named `demo_txt` , and assigns it a listener object, `txtListener` , that resp to the *onChanged( )* event. For comparison, the equivalent callback function approach is also shown. To see the code wor copy it into a new movie, export the movie to Test Movie mode, then enter text in the text field.

```
// Make the text field.
this.createTextField("demo_txt", 1, 10, 10, 200, 20);
// Display some text in the field.
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

```
demo_txt.text = "type text here";
// Make the field user-editable.
demo_txt.type = "input";
// Create an anonymous object to act as the listener.
txtListener = new Object();
// Add the onChanged function to the listener. Notice that the function
// is passed the name of the field that triggered the event.
txtListener.onChanged = function (tfObject) {
  trace(tfObject + " changed. Listener invoked.");
}
// Now add the listener to demo_txt.
demo_txt.addListener(txtListener);
// For comparison, here's how we can do the same thing with a callback funct
demo_txt.onChanged = function () {
  trace(this + " changed. Callback invoked.");
}
```

See Also

TextField.onChanged( ) , *TextField.onScroller( )* , *TextField.removeListener( )* ; Chapter 10

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Chapter 18. ActionScript Language Reference

TextField._alpha Property Flash 6

opacity percentage: 0 is transparent; 100 is opaque *read/write*

theField._alpha

Description

The floating-point _alpha property specifies the opacity (or, conversely, the transparency) of *theField* as a percentage—0 is completely transparent, whereas 100 is completely opaque. The text, background, and border of *theField* are all affected. The _alpha property works only when *theField*'s font is embedded and exported with the movie. A font can be embedded at authoring time with the Property inspector's Character button or at runtime with the embedFonts property. For important details on font embedding, see *TextField*.embedFonts.

Example

Assuming its font has been exported with the movie, the following code makes theField_txt 50% transparent:

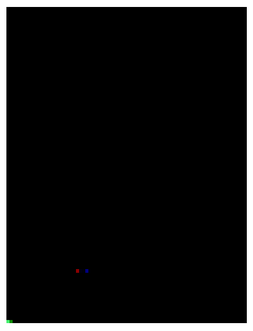
```
theField_txt.embedFonts = true;
theField_txt._alpha = 50;
```

See Also

TextField._visible, *TextField*.embedFonts

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir



Chapter 18. ActionScript Language Reference

TextField.autoSize Property — Flash 6

matches the text field's bounding rectangle size to its text — *read/write*

```
theField.autoSize
```

Description

The string `autoSize` property can make a text field's bounding rectangle match the size of the text it contains. Use `autoSize` to ensure that a text field is large enough to display its assigned text or to allow a text field to change size dynamically when its content changes (due to user input or reassignment of the `text` property). For example, here we create a *TextField*, `title_txt`, that contains the text "Contact Information". We don't know the size of the text, so we can't set the width of `title_txt` accurately:

```
// The width and height of 80 and 20 are just a guess...
this.createTextField("title_txt", 1, 0, 0, 80, 20);
title_txt.text = "Contact Information";
title_txt.border = true;
```

By setting the `autoSize` property to "left", we guarantee that the words "Contact Information" will be visible:

```
title_txt.autoSize = "left";
```

The `autoSize` property has four possible values: "none" (the default), "left", "right", and "center". Generally, these dictate the direction in which `theField` should expand or contract to fit its text, where "none" means don't resize, "left" means resize right and bottom, "right" means resize left and bottom, and "center" means resize evenly on both sides and bottom. When assigned a Boolean value, `autoSize` interprets `false` as a synonym for "none" and `true` as a synonym for "left". However, `autoSize` has important interactions with the `multiline` and `wordWrap` properties. Table 18-22 shows the behavior of `autoSize` in relation to `multiline` and `wordWrap`; read the table left to right to see which settings cause text field edges to resize. Note that the default values for both `multiline` and `wordWrap` are `false`.

Table 18-22. The autoSize property behavior

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

autoSize	multiline	wordWrap	Resize right?	Resize left?	Resize bottom?
`false` or `"none"`	`true` or `false`	`true` or `false`	no	no	no
`true` or `"left"`	`false`	`false`	yes	no	no
`true` or `"left"`	`true`	`false`	yes	no	yes (only at linefeeds)
`true` or `"left"`	`false`	`true`	no	no	yes (only at text overflow)
`true` or `"left"`	`true`	`true`	no	no	yes (at text overflow and linefeeds)
`"right"`	`false`	`false`	no	yes	no
`"right"`	`true`	`false`	no	yes	yes (only at linefeeds)
`"right"`	`false`	`true`	no	no	yes (only at text overflow)
`"right"`	`true`	`true`	no	no	yes (at text overflow and linefeeds)
`"center"`	`false`	`false`	yes	yes	no
`"center"`	`true`	`false`	yes	yes	yes (only at linefeeds)
`"center"`	`false`	`true`	no	no	yes (only at text overflow)
`"center"`	`true`	`true`	no	no	yes (at text overflow and linefeeds)

The `autoSize` property overrides any absolute sizes specified by *createTextField( )*, `_height`, or `_width`.

Example

The following code causes `title_txt`'s bounding rectangle to grow or shrink on both the left and right sides as text is added to it or removed from it:

```
this.createTextField("title_txt", 1, 0, 0, 0, 0);
title_txt.text = "Title Goes Here";
title_txt.border = true;
title_txt.autoSize = "center";
```

The following code gives `paragraph_txt` a fixed width of 100 and an expandable bottom border:

```
this.createTextField("paragraph_txt", 1, 0, 0, 100, 0);
paragraph_txt.text = "The following sentence is a pangram.\n\n"
                   + "The quick brown fox jumps over a lazy dog.";
paragraph_txt.border = true;
paragraph_txt.multiline = true;
paragraph_txt.wordWrap = true;
paragraph_txt.autoSize = "left";
```

See Also

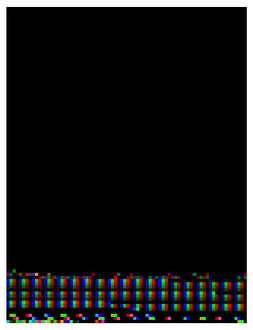
TextField._height, *TextField*.multiline, *TextField*._width, *TextField*.wordWrap

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir



Chapter 18. ActionScript Language Reference

TextField.background Property Flash 5

Boolean; turns text field background on and offFlash 6; accessible only via authoring tool in Flash 4 and ***read/write***

theField.background

Description

The Boolean background property specifies whether to show a colored rectangle behind a field's text (true) or to make the bounding rectangle behind the field transparent (false, the default). The rectangle's color is specified by the backgroundColor property (defaults to white). To change the size of the rectangle, use the _width and _height properties. To add a border to the rectangle, use the border and borderColor properties.

To produce a more elaborate background for a text field, set background and border to false and manually draw a shape behind the text field.

Example

Put a white background on theField_txt:

```
this.createTextField("theField_txt", 1, 0, 0, 80, 20);
theField_txt.text = "hello world";
theField_txt.background = true;
```

Now make the white background red:

```
theField_txt.backgroundColor = 0xFF0000;
```

See Also

TextField.backgroundColor, *TextField*.border, *TextField*.borderColor, *TextField*._width, *TextField*._height

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Chapter 18. ActionScript Language Reference

TextField.backgroundColor Property — Flash 6

RGB number specifying the text field's background color — *read/write*

`theField.backgroundColor`

Description

The integer `backgroundColor` property sets the color of a text field's background. Colors are specified as a single number, most commonly a six-digit hexadecimal value of the form `0x`*`RRGGBB`*, where *`RR`*, *`GG`*, and *`BB`* are two-digit numbers between `00` and `FF`, representing the Red, Green, and Blue components. For example, the RGB triplet (R: 51, G: 51, B: 102) is equivalent to the hexadecimal value:

```
0x333366
```

The default value is white (`0xFFFFFF`). For more information on RGB triplet notation, see the *Color* class.

The `backgroundColor` property has a visible effect only when the `background` property is `true`; when `background` is `false`, no background is shown, even if `backgroundColor` is set. For example, the following code tries to set the background color of `theField_txt` to blue, but the color does not appear because `background` is not yet set:

```
this.createTextField("theField_txt", 1, 0, 0, 200, 20);
theField_txt.text = "this text has a blue background";
theField_txt.backgroundColor = 0x0000FF;
```

To make the blue background appear, we must add the following line:

```
theField_txt.background = true;
```

Setting the value of `background` does not alter the value of `backgroundColor`.

See Also

`TextField``.background`, *`TextField`*`.border`, *`TextField`*`.borderColor`, the *Color* class

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Chapter 18. ActionScript Language Reference

TextField.border Property — Flash 5

Boolean; turns text field border on and off — **Flash 6; accessible only via authoring tool in Flash 4 and** — ***read/write***

theField.border

Description

The Boolean `border` property specifies whether to show an outline around *`theField`*'s bounding rectangle (`true`) or not (`false`). The outline color is specified by `borderColor` (defaults to black). To change the size of the rectangle, use the `_width` and `_height` properties. To add a background color to the rectangle, use the `background` and `backgroundColor` properties.

To produce a more elaborate background for a text field, set `background` and `border` to `false`, and manually draw a shape behind the text field.

Example

Put a black border on `theField_txt`:

```
this.createTextField("theField_txt", 1, 0, 0, 80, 20);
theField_txt.border = true;
theField_txt.text = "hello world";
```

Now make the border blue:

```
theField_txt.borderColor = 0x0000FF;
```

See Also

TextField.background, *TextField*.backgroundColor, *TextField*.borderColor, *TextField*. _height, *TextField*._width

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Chapter 18. ActionScript Language Reference

TextField.borderColor Property — Flash 6

RGB number specifying the text field's border color — *read/write*

theField.borderColor

Description

The integer borderColor property sets the color of the outline around a text field's bounding rectangle. Colors are specified as a single number, most commonly a six-digit hexadecimal value of the form 0x*RRGGBB*, where *RR*, *GG*, and *BB* are two-digit numbers between 00 and FF, representing the Red, Green, and Blue components. For example, the RGB triplet (R:51, G:51, B:102) is equivalent to the hexadecimal value:

```
0x333366
```

The default value is black (0x000000). For more information on RGB triplet notation, see the *Color* class.

The borderColor property has a visible effect only when the border property is true; when border is false, no border is shown, even if borderColor is set. For example, the following code tries to set the border color of theField_txt to green, but the color does not appear because border is not yet set:

```
this.createTextField("theField_txt", 1, 0, 0, 200, 20);
theField_txt.text = "this text has a green border";
theField_txt.borderColor = 0x00FF00;
```

To make the green border appear, we must add the following line:

```
theField_txt.border = true;
```

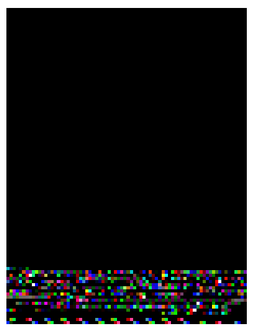
Setting the value of border does not alter the value of borderColor.

See Also

TextField.background, *TextField*.backgroundColor, *TextField*.border, the *Color* class

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir



Chapter 18. ActionScript Language Reference

TextField.bottomScroll Property — Flash 6

the 1-relative index of the text field's lowest visible line — ***read-only***

theField.bottomScroll

Description

The integer bottomScrollproperty represents the 1-relative line number of the bottommost line currently displayed in a text field.

When the text in a text field spans more lines than can be accommodated by the physical viewable region of the field, extra lines of text are hidden but are still part of the text field. The top line is number 1, and line numbers increase for every line in the text field, including those that exceed the viewable boundaries of the field. See Figure 18-6 under *TextField*.scroll, which shows a sample text field's line index values.

When a text field contains more lines than it can display at once, we can determine the range of lines displayed by using the scroll and bottomScroll properties. The maxScroll property returns the index of the top visible line when the field is scrolled as far as possible, but it doesn't tell us the total number of lines in the field. Though there is no separate property for this, we can determine the index of the last line by temporarily setting scroll to maxScroll and checking bottomScroll, as follows:

```
theField_txt.oldScroll = theField_txt.scroll;
theField_txt.scroll = theField_txt.maxscroll;
trace("Number of lines in text field is " + theField_txt.bottomScroll);
theField_txt.scroll = theField_txt.oldScroll;
```

Reader Exercise: Create a custom property, maxBottomScroll, that stores the number of lines in a field. Write a *getMaxBottomScroll( )* function to retrieve it.

Example

We can calculate the number of lines currently displayed in a text field (sometimes referred to as the field's *page size*) by subtracting scroll - 1 from bottomScroll. The following code adds a *getPageSize( )* method to all *TextField* objects; when invoked on a *TextField* instance, the method returns the number of viewable lines in that field:

```
TextField.prototype.getPageSize = function () {
  var pageSize = this.bottomScroll - (this.scroll - 1);
  trace("Page size is " + pageSize);
  return pageSize;
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

```
}
```

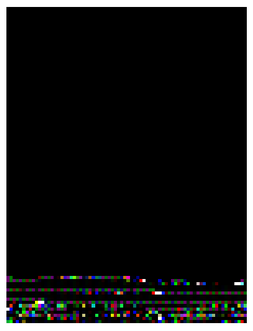
The page size of a text field can be used to scale a scroll handle proportionately in a corresponding scroll bar. Note that the number of displayable lines in a field can vary, depending on the font sizes of the currently displayed text.

See Also

`TextField``.maxscroll`, *`TextField`*`.scroll`; the *FScrollBar* component in Appendix E

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir



Chapter 18. ActionScript Language Reference

TextField.condenseWhite Property Flash 6 (undocumented)

specifies whether HTML whitespace should be condensed *read/write*

theField.condenseWhite

Description

When set to `true`, the Boolean `condenseWhite` property causes a field's HTML text to be condensed, removing all excess whitespace characters (tab, line break, space, etc.). Hence, the field's HTML rendering mirrors a typical web browser—multiple spaces behave as a single space between words and the newline character "`\n`" no longer serves as a line break; the `<P>` and `<BR>` tags must be used instead. In this mode, HTML text parsing behaves like Flash's XML parser when the *XML*`.ignoreWhite` property is set to `true`.

The `condenseWhite` property allows you to use existing HTML text in Flash without removing excess whitespace manually. The default value of `condenseWhite` is `false`. Even when set to `true`, it has no effect unless the field's `html` property is also `true`.

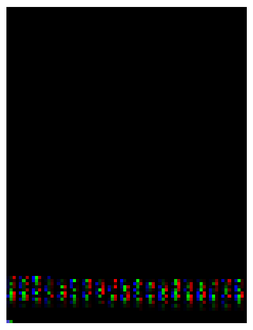
The `condenseWhite` property is an undocumented feature of the *TextField* object, but you can rely on it being supported in future versions as well. The overly cautious developer can manually remove whitespace from the HTML text to make it appear the same in Flash as it does in a web browser.

See Also

TextField.html, *TextField*.htmlText, *XML*.ignoreWhite

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir



Chapter 18. ActionScript Language Reference

TextField.embedFonts Property Flash 6

render text using fonts exported with the movie ***read/write***

theField.embedFonts

Description

The Boolean `embedFonts` property specifies whether a field's text is rendered using *device fonts* on the user's system (`false`) or fonts embedded in the *.swf* file (`true`). When `embedFonts` is `false`, text appears without antialiasing and missing fonts are replaced by the system's default font (usually some variation of Times New Roman). When `embedFonts` is `true` and the field's fonts are exported with the *.swf* file, text appears antialiased. When `embedFonts` is `true` but one or more of the field's fonts is missing from the *.swf* file, text in the missing fonts does not appear at all. Embedding a complete Roman font typically adds 20-30 KB to a movie (Asian fonts can be much larger). Using embedded fonts with sizes smaller than 10-point is not recommended, because antialiased text becomes unreadable below 10-point in most fonts.

> The contents of a text field that is rotated or masked with an author-time mask layer will not show up on screen unless its font is embedded. However, as of Flash Player 6.0.40.0, *MovieClip.setMask( )* can be used to apply a rectangular mask to a text field at runtime without embedded fonts. See *MovieClip.setMask( )* for details.

To export (i.e., to embed) a font with a *.swf* file, we must do one of the following:

- Create a new font symbol in the movie's Library.
- Manually add a dummy text field at authoring time and set its Character options in the Property inspector.

The font symbol approach always exports an entire font, while the text field approach allows us to export only part of a font, reducing file size. To export a subset of a font with a font symbol, we must use font-making software such as Macromedia Fontographer to create a custom font as a subset of the original font.

To export Arial Bold using a font symbol:

1. Select Window Library.
2. From the pop-up Options menu in the upper-right corner of the panel, select New Font. The Font Symbol Properties dialog box appears.
3. Under Font, select Arial.
4. Under Style, select Bold.
5. Under Name, type **`ArialBold`** (this is a cosmetic name, used in the Library only).
6. In the Library, select the `ArialBold` font symbol.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

7. From the pop-up Options menu, select Linkage.
8. In the Symbol Linkage Properties dialog box, under Linkage, select Export For ActionScript.
9. In the Identifier box, type **`ArialBold`**. This name will be used as the value of a *TextFormat* object's `font` property.

To export any font using a dummy text field:

1. Select the Text tool.
2. Draw a rectangle on stage.
3. In the Property inspector, choose the desired font. The font's actual name will be used as the value of a *TextFormat* object's `font` property.
4. In the Property inspector, click the Character button and choose the character outlines to export.

Every variation of a font style must be embedded individually. If we use Courier New in bold, italic, and bold italic in a text field, then we must embed all three font variations or the text will not display correctly. Underline is not considered a font variation, nor is font size or color.

Setting `embedFonts` to `true` does not cause any fonts to download with the movie; it merely indicates that ***`theField`*** should be rendered with exported fonts if they are available. The `embedFonts` property must be set separately for each text field that uses a particular font, even if multiple text fields use the same font. However, file size is not affected when multiple text fields embed the same font—only one copy of the font is downloaded with the movie.

To specify the font for a text field, use a *TextFormat* object and the *TextField.setTextFormat( )* and *TextField.setNewTextFormat( )* methods. To retrieve a list of fonts available on the user's system, use *TextField.getFontList( )*.

On operating systems that support Unicode (e.g., Windows XP and Mac OS X), when `embedFonts` is `false` and a character is needed that is not available in the specified font, Flash will automatically search the system for a font that can represent the character. If a suitable font is found, it will be used until the next missing character is encountered, at which point Flash will again search for a suitable font.

Bugs

Theoretically, to apply the same `embedFonts` value to all text fields in a movie, we would use this syntax:

```
TextField.prototype.embedFonts = true;
```

However, in Flash 6, built-in properties set on `TextField.prototype` are not inherited by text field instances.

Example

Assuming the Verdana font is exported, the following code causes `theField_txt` to be rendered in antialiased Verdana:

```
// Make the text field
this.createTextField("theField_txt", 1, 0, 0, 150, 40);
// Tell the field to use embedded fonts
theField_txt.embedFonts = true;
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

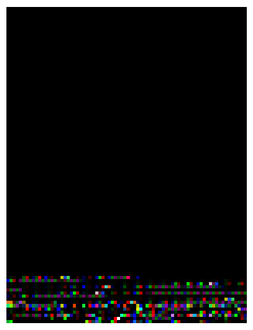
```
// Assign some text to the field
theField_txt.text = "How are you?"
// Create and apply a text format with a font of Verdana
defaultFormat = new TextFormat();
defaultFormat.font = "Verdana";
theField_txt.setTextFormat(defaultFormat);
```

See Also

TextField.getFontList( ), *TextField.setNewTextFormat( )*, *TextField.setTextFormat( )*, `TextFormat.font`

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir



TextField.getDepth( ) Method Flash 6

returns the position of the text field in the content stack

```
theField.getDepth()
```

Returns

An integer depth level.

Description

Text fields, movie clips, and buttons are all placed in the Player in an order that governs how objects stack visually. An internal depth property indicates each object's order in the *content stack*. The *getDepth( )* method tells us a field's integer depth position. Each depth level can have only one occupant, whether the occupant is a text field, button, or movie clip. Objects on higher depths appear in front of those with lower depths. For text fields created at runtime with *MovieClip.createTextField( )*, the return value of *getDepth( )* is the depth assigned to the method's `depth` parameter. For example:

```
// Sets depth to 13
this.createTextField("theField_txt", 13, 0, 0, 150, 20);
trace(theField_txt.getDepth());  // Displays: 13
```

For text fields created manually at authoring time, *getDepth( )* returns a depth in the range of -16383 to -1 (inclusive). Unlike the *MovieClip* class, *TextField* has no *swapDepths( )* method, so each text field's depth is fixed once it has been assigned.

For much more information about depths, see Section 13.4 in Chapter 13.

See Also

MovieClip.createTextField( ), *MovieClip.getDepth( )*, *MovieClip.swapDepths( )*; "Movie and Instance Stacking Order," in Chapter 13

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Chapter 18. ActionScript Language Reference

TextField.getFontList( ) Class Method Flash 6

returns an array of fonts on the user's system

```
TextField.getFontList()
```

Returns

An array of font names as strings.

Description

The *getFontList( )* class-level method tells us what fonts are installed on the system playing a Flash movie. We can use an the fonts listed as a device font for a text field, provided that the field has `embedFonts` set to `false` . To assign a fon a text field, use the `font` property of a *TextFormat* object and the *TextField.setTextFormat( )* method.

Only families of fonts are listed by *getFontList( )* , as would appear in the text field Property inspector's font selector. Variations, such as bold or italic versions of a font, are not listed. To use a bold or italic variation of a font, set the `bold` `italic` property of a *TextFormat* object, in addition to the `font` property, before applying it using *TextField.setTextFormat( )* or *TextField.setNewTextFormat( )* .

Typical uses for *getFontList( )* include selecting a fallback font automatically (see the following Example) and letting the select a font.

Usage

Note that *getFontList( )* is called directly as a method of the *TextField* class, not through an instance of the class:

```
someTextField_txt.getFontList();  // Wrong way...won't work.
TextField.getFontList()           // Right way.
```

Bugs

In Flash 6, the *getFontList( )* method does not return names of fonts exported with a movie.

Example

The function *assignFont( )* takes an array of font names and returns the first one that is found on the user's system. If none found, *assignFont( )* returns `null` :

```
this.createTextField("theField_txt", 1, 0, 0, 150, 40);
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

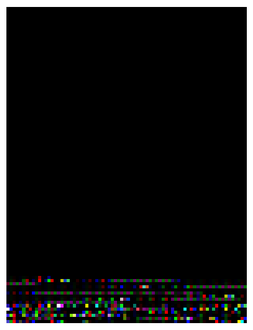
```
theField_txt.text = "Flash is fun!";
defaultFormat = new TextFormat();
defaultFormat.font = assignFont(["ZapfChancery", "Verdana", "Arial", "_sans"
theField_txt.setTextFormat(defaultFormat);
function assignFont (fontList) {
  var systemFonts = TextField.getFontList();
  for (var i = 0; i < fontList.length; i++) {
    for (var j = 0; j < systemFonts.length; j++) {
      if (fontList[i] =  = systemFonts[j]) {
         return fontList[i];
      }
    }
  }
  return null;
}
```

See Also

TextField .embedFonts , *TextField.setTextFormat( )* , *TextFormat* .bold , *TextFormat* .font , *TextFormat* .italic , the *TextFormat* class

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir



Chapter 18. ActionScript Language Reference

TextField.getNewTextFormat( ) Method Flash 6

retrieves the default TextFormat object for the text field

```
theField.getNewTextFormat()
```

Returns

A *TextFormat* object describing the default format for any characters added to `theField`.

Description

The *getNewTextFormat( )* method returns a *TextFormat* object whose properties describe the format applied to any new characters added to `theField`. For a detailed description of *TextFormat* objects, see the *TextFormat* class.

Any text added to a field via `TextField.text`, *TextField.replaceSel( )*, or by the user is formatted according to the settings of the object returned by *getNewTextFormat( )*. For a new text field, the default *TextFormat* object has the property values shown in Table 18-25 under the Description heading for the *TextFormat* class entry.

The default values can be changed with *setNewTextFormat( )*. However, *setNewTextFormat( )* does not format characters already in the field; it merely sets the format for any future characters that are added. To reformat characters already in a text field, use *setTextFormat( )*. To examine the format of characters already in a text field, use *getTextFormat( )*.

Usage

When the insertion point is placed in a text field by the user or with *Selection.setSelection( )*, the field's new text format is set to match the character to the insertion point's right (or left, in the case of the last character). The return value of *getNewTextFormat( )* will reflect the format of that character. See the Example under *TextField.setNewTextFormat( )* for code that forces the new text format not to adopt the existing text's format.

Example

The following code checks the default font size for a field named `numbers_txt` and then increases that size by ten points:

```
// Create a text field
this.createTextField("numbers_txt", 1, 0, 0, 150, 40);
// Assign the field some text to display
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

```
numbers_txt.text = "one two three four";
// Retrieve the new text format object for numbers_txt
var numbersNewTextFormat = numbers_txt.getNewTextFormat();
// Check the default font point size for new text
trace(numbersNewTextFormat.size);  // Displays: 12
// Add ten points and apply the new default settings
numbersNewTextFormat.size += 10;
numbers_txt.setNewTextFormat(numbersNewTextFormat);
// Now insert some text to see the change
numbers_txt.text = "five six seven";
```

See Also

TextField.getTextFormat( ), *TextField.setNewTextFormat( )*, *TextField.setTextFormat( )*, the *TextFormat* class

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Chapter 18. ActionScript Language Reference

TextField.getTextFormat( ) Method Flash 6

retrieves a TextFormat object for specified characters

```
theField.getTextFormat()
theField.getTextFormat(beginIndex)
theField.getTextFormat(beginIndex, endIndex)
```

Arguments

beginIndex

The index of the first character whose formatting is described by the returned *TextFormat* object. It is optional, but must be a nonnegative integer if it is included. If it is omitted, the entire field is used.

endIndex

The index following the last character whose formatting is described by the returned *TextFormat* object. It is optional, but it must be a nonnegative integer if it is included.

Returns

A *TextFormat* object describing the formatting of the specified characters.

Description

The *getTextFormat( )* method lets us examine the existing formatting of one or more characters already in a field. It return *TextFormat* object whose properties describe the formatting of the specified characters. For a detailed description of *TextFormat* objects, see the *TextFormat* class.

When *getTextFormat( )* is invoked with one integer argument, or if `endIndex` is equal to `beginIndex + 1`, the returned *TextFormat* object reflects the formatting for the single character at `beginIndex`. For example, here we app a format to the first four characters of a text field, and then we check the font of the first character:

```
// Create the text field
this.createTextField("theField_txt", 1, 0, 0, 150, 40);
// Assign the field some text for display
theField_txt.text = "What time is it?";
// Create a TextFormat object with a font property of "Arial"
sansFormat = new TextFormat("Arial");
// Apply the format to the word 'What', characters 0 to 3 (inclusive)
theField_txt.setTextFormat(0, 4, sansFormat);
// Retrieve a TextFormat object for the first character
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

```
firstCharFormat = theField_txt.getTextFormat(0);
// Check the font property
trace(firstCharFormat.font);  // Displays: Arial
```

When *getTextFormat( )* is invoked with two integer arguments, the returned *TextFormat* object represents the formatting f the span of characters from `beginIndex` to `endIndex - 1` . And when *getTextFormat( )* is invoked with no arguments, the returned *TextFormat* object represents the formatting for all the characters in the field.

If a specific format (e.g., font, bold, or italic) is not the same for all characters in a specified span, the corresponding property of the *TextFormat* object for that span will be `null` . Continuing with our example, if we retrieve a *TextFormat* object for the entire text of `theField_txt` , we find that properties shared by all characters return non-`null` values:

```
// Retrieve a TextFormat object for all characters in theField_txt
allCharsFormat = theField_txt.getTextFormat();
// Now check whether all the characters are bold
trace(allCharsFormat.bold);  // Displays: false
```

But properties that vary between characters return `null` :

```
// Check the font for all characters
trace(allCharsFormat.font);  // Displays: null (if font is not uniform)
```

The first four characters in `theField_txt` have a different font than the remaining characters, so no single `font` property value can accurately describe the entire span and `null` is returned.

Note that changes to a *TextFormat* object returned by *getTextFormat( )* do not have any effect on the text of `theField` until the object is applied with *setTextFormat( )* . For example, on its own, the following assignment of the `font` propert has no effect on `theField_txt` :

```
allCharsFormat.font = "Courier New";
```

But when we add a call to *setTextFormat( )* , the change is applied:

```
theField_txt.setTextFormat(allCharsFormat);  // Applies the "Courier New" fo
                                             // to the whole text field
```

Don't confuse *getNewTextFormat( )* with *getTextFormat( )* ; the former retrieves the format used for text that will be adde to `theField` in the future, while the latter retrieves the format for characters already in `theField` .

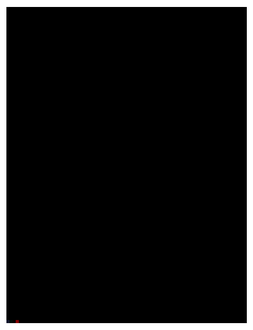
For a detailed description of *TextFormat* objects, see the *TextFormat* class entry.

See Also

TextField.getNewTextFormat( ) , *TextField.setNewTextFormat( )* , *TextField.setTextFormat( )* , the *TextFormat* class

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir



Chapter 18. ActionScript Language Reference

TextField._height Property Flash 6

height of the text field's bounding box, in pixels ***read/write***

theField._height

Description

Every text field defines a rectangular region, known as its *bounding box*, inside which text is displayed. The floating-point _height property is a nonnegative number specifying the height of a field's bounding box, in pixels. By changing _height, we resize a field's bounding box, allowing it to display more or less text. Changing a field's _height doesn't change the height of the text it contains; to change text size, set the *TextFormat*.size or *TextField*._yscale properties. To check the height of a field's text, use textHeight. Note that as of Flash 6, the autoSize property can be used to force a text field's bounding box to fit its text.

Changes made to a clip's height in the authoring tool or via _yscale are reflected by _height. To apply a stroke or a fill to a field's bounding box, use the border and background properties.

Example

```
// Create a 10-pixel-high text field
// (with a border so we can see its bounding box)
this.createTextField("theField_txt", 1, 0, 0, 150, 10);
theField_txt.border = true;
// Assign the field some text for display
theField_txt.text = "Almost time for breakfast.";
// Make theField_txt 20 pixels high
theField_txt._height = 20;
// Add 15 pixels to theField_txt's height
theField_txt._height += 15;
```

See Also

TextField.autoSize, *TextField*.textHeight, *TextField*._width, *TextField*._yscale, the *TextFormat* class

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Chapter 18. ActionScript Language Reference

TextField.hscroll Property — Flash 6

a text field's horizontal scroll position, in pixels ***read/write***

theField.hscroll

Description

When a field's text is wider or higher than can be accommodated by the physical viewable region of its bounding box, exc
hidden from view. To bring the text into view, we can scroll the field horizontally or vertically using the `hscroll` and `s`
properties. The integer `hscroll` property specifies the number of pixels a field's text is scrolled to the left. The default,
indicates that the text is not scrolled horizontally (the first character in the field is flush with the left border of the field's b
box). If a field's text can be fully displayed without any horizontal cutoff, `hscroll` remains at 0 and cannot be set. But i
text cannot be fully displayed horizontally in the allotted space, setting `hscroll` to a positive integer offsets the text by
specified number of pixels to the left, up to the maximum set by `maxhscroll` . When more than one line is displayed i
lines are moved simultaneously. Scrolling caused by user input is reflected by `hscroll` . Note that `hscroll` does not
one character at a time; it scrolls the field one pixel at a time. This contrasts with `scroll` , which moves text one line at
one pixel at a time.

The `hscroll` property typically is used to create horizontal scrolling interfaces for a text field, or for "stock ticker" styl
display, as shown in the following Example.

Example

Example 18-5 shows a simple but flexible text ticker that uses the `hscroll` property to scroll its text horizontally in an
cycle.

Example 18-5. Text ticker

```
// Create a text field (with a border so we can see its bounding box)
this.createTextField("ticker_txt", 1, 0, 0, 100, 20);
ticker_txt.border = true;
// Set speed of ticker_txt (pixels per 20 ms)
ticker_txt.speed = 1;
// Assign the field some text for display
ticker_txt.text = "Here is the latest news from moock.org: "
                + "Colin drinks a cup of tea!";
// Assign a buffer of spaces so the text flows in and out of the text field
// First, check the size of a space character in this field's format
ticker_txt.spaceSize = ticker_txt.getNewTextFormat().getTextExtent(" ").widt
// Now calculate how many spaces we need to fill the field
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

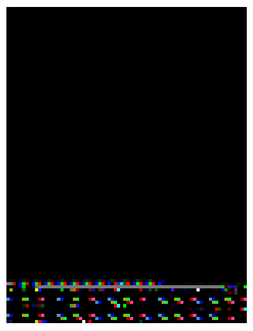
```
ticker_txt.spacesRequired = Math.ceil(ticker_txt._width / ticker_txt.spaceSi
// Create a string of spaces
for (var i = 0; i < ticker_txt.spacesRequired; i++) {
  ticker_txt.spacebuffer += " ";
}
// Add the spaces to the front and end of the text
ticker_txt.text = ticker_txt.spacebuffer + ticker_txt.text + ticker_txt.spac
// Create a function to scroll the text
ticker_txt.hscrollInterval = function () {
  // If we're at the end of the text...
  if (this.hscroll =  = this.maxhscroll) {
    // ...start again
    this.hscroll = 0;
  }
  // Scroll the text by the specified amount
  this.hscroll += this.speed;
}
// Call our text scroller every 20 milliseconds
setInterval(ticker_txt, "hscrollInterval", 20);
```

See Also

setInterval( ) , *TextField.getNewTextFormat( )* , *TextField* .maxhscroll , *TextField* .scroll , *TextF* .width , *TextField* .wordWrap , *TextField* ._yscale , *TextFormat.getTextExtent( )* ; the *FScrollBar* c in Appendix G

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir



Chapter 18. ActionScript Language Reference

TextField.html Property — Flash 5

Boolean; enables or disables HTML displayFlash 6; accessible via authoring tool only in ***read/write***

theField.html

Description

The Boolean `html` property indicates whether a field should render the contents of its `htmlText` property as formatted HTML (`true`) or as plain text (`false`). It defaults to `false`. To have an effect, `html` must be `true` before `htmlText` is assigned. In Flash 5, HTML provided the only means of achieving complex text formatting in a text field; as of Flash 6, the *TextFormat* class offers object-oriented formatting control.

For more information, see *TextField*.htmlText.

Example

The following code creates a text field that displays italicized text:

```
this.createTextField("theField_txt", 1, 0, 0, 200, 20);
theField_txt.html = true;
theField_txt.htmlText = '<I>This will display italicized text.</I>';
```

The text field displays:

> *This will display italicized text.*

If line 2 were omitted, the text field would display:

> <I>This will display italicized text.</I>

See Also

TextField.condenseWhite, *TextField*.htmlText, *TextField*.text, the *TextFormat* class; Appendix E

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Chapter 18. ActionScript Language Reference

TextField.htmlText Property — Flash 6

HTML source code to render in the text field — *read/write*

theField.htmlText

Description

Like the `text` property, `htmlText` is used to specify text for on-screen display but with the added feature that if the `h` property is `true` , HTML tags in `htmlText` are interpreted as formatting instructions. To display formatted HTML in field, first set `html` to `true` , and then assign HTML source code to `htmlText` :

```
this.createTextField("theField_txt", 1, 0, 0, 200, 20);
theField_txt.html = true;
theField_txt.htmlText = '<B>This will display bold text.</B>';
```

The set of supported tags is limited, compared to a web browser, but it still provides good basic text control. See Appendi for details.

> Assigning a value to `htmlText` overwrites the value of `text` , and vice versa. In most cases, if you are using HTML text, set the `html` property to `true` and access the field's text using the `htmlText` property. If you are using non-HTML text, set the `html` property to `false` and access the field's text using the `text` property.

When `html` is `false` , HTML tags in text assigned to `htmlText` show up as regular text, and `htmlText` behaves i same way as the `text` property. HTML tags assigned directly to `text` are never interpreted as HTML; they are always displayed verbatim.

However, when `html` is `true` and HTML tags are assigned to `htmlText` , the value of `text` will be that of `htmlT` but with all HTML tags stripped out. For example, here we create an HTML-enabled text field and assign some HTML to `htmlText` property for display:

```
this.createTextField("theField_txt", 1, 0, 0, 200, 20);
theField_txt.html = true;
theField_txt.htmlText = '<P ALIGN="LEFT"><FONT FACE="Times New Roman" SIZE="
                        + 'COLOR="#000000">This field contains <B>HTML!</B>'
                        + '</FONT></P>';
```

After the assignment, `htmlText` has the value:

```
<P ALIGN="LEFT"><FONT FACE="Times New Roman" SIZE="12" COLOR="#000000">
This field contains <B>HTML!</B></FONT></P>
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

But `text` has the value:

```
This field contains HTML!
```

Conversely, as long as `html` is `true` , Flash automatically adds HTML markup to `htmlText` if the HTML source ass to `htmlText` does not fully describe the text field's formatting. For example, here we set `htmlText` without using an or `<FONT>` tags:

```
theField_txt.htmlText = "This field contains <B>HTML!</B>";
```

When we examine the value of `theField_txt.htmlText` , we find that `<P>` and `<FONT>` tags have been added:

```
trace(theField_txt.htmlText);
// Displays:
<P ALIGN="LEFT"><FONT FACE="Times New Roman" SIZE="12" COLOR="#000000">
This field contains <B>HTML!</B></FONT></P>
```

Meanwhile, `text` has the value:

```
This field contains HTML!
```

When *`theField`* `.html` is `true` and *`theField`* is formatted with *setTextFormat( )* , `htmlText` represents the formatting with HTML tags. For example:

```
this.createTextField("theField_txt", 1, 0, 0, 200, 30);
theField_txt.html = true;
theField_txt.text = "Stretch often while you work.";
// Create a TextFormat object that specifies bold text.
boldFormat = new TextFormat();
boldFormat.bold = true;
boldFormat.size = 20;
// Apply the format to the word "often", characters 8 to 12 (inclusive).
theField_txt.setTextFormat(8, 13, boldFormat);
// Check the value of htmlText. Note the <B> tags around the word "often".
trace(theField_txt.htmlText);
// Displays:
<P ALIGN="LEFT"><FONT FACE="Times New Roman" SIZE="12" COLOR="#000000">
Stretch <FONT SIZE="20"><B>often</B></FONT> while you work.</FONT></P>
```

Note, however, that the new text format for *`theField`* (as set by *setNewTextFormat( )*) is not used by `htmlText` , a demonstrated in the following code:

```
this.createTextField("theField_txt", 1, 0, 0, 250, 100);
theField_txt.html = true;
// Set the new text format to be bold, 20 pt.
tformat = theField_txt.getNewTextFormat();
tformat.size = 20;
tformat.bold = true;
theField_txt.setNewTextFormat(tformat);
// This text, assigned to htmlText, does not take the formatting
theField_txt.htmlText = '<P>This text is size 12, not size 20.</P>';
// This text, assigned to text, will be size 20, bold
theField_txt.setNewTextFormat(tformat);
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

```
theField_txt.text = 'This text is size 20, bold.';
```

Table 18-23 shows the effect of setting the `htmlText` property under different scenarios.

Table 18-23. Display when setting the htmlText property

html	Assigned to htmlText	htmlText contents	text contents	On-scr displa
false	<I>Italicized text</I>	<I>Italicized text</I>	<I>Italicized text</I>	<I>Italiciz text</I>
false	Plain text	Plain text	Plain text	Plain text
true	<I>Italicized text</I>	<P ALIGN="LEFT"><FONT FACE="Times New Roman" SIZE="12" COLOR="#000000"><I>Italicized text</I></FONT></P>	Italicized text	*Italicized t*
true	Plain text	<P ALIGN="LEFT"><FONT FACE="Times New Roman" SIZE="12" COLOR="#000000">Plain text</FONT></P>	Plain text	Plain text

Take heed that successive assignments to `htmlText` and `text` overwrite each other. Regardless of the value of *`theF`* `.html` , assigning a new value to `text` overwrites the value of `htmlText` , and vice versa. Successive *concatenation* reassignments), in contrast, do not overwrite each other:

```
theField_txt.htmlText = "<B>hello</B>";
theField_txt.text += " world";
// theField_txt.htmlText is now:
// <P ALIGN="LEFT"><FONT FACE="Times New Roman" SIZE="12"
// COLOR="#000000">hello world</FONT></P>
```

But concatenating `text` to `htmlText` can cause problems; when we assigned a value to `text` , Flash removed our `<P>` assigned to `htmlText` because assigning a new value to `text` removes all custom formatting in the field. Further, eacl concatenation or assignment to `htmlText` causes the insertion of a new paragraph tag (`<P>`). Hence, mixing `text` an `htmlText` assignments generally is not recommended. Each assignment made to `htmlText` should comprise an entir paragraph; otherwise, any subsequent concatenation will force an undesired line break.

To include HTML source code in an HTML-enabled text field, convert the characters <, >, &, ", and ' to their equivalent H entity values: `<` , `>` , `&` , `"` , and `'` , as discussed at: http://moock.org/asdg/technotes/sourceInHtmlField.

To display HTML tags in a non-HTML text field, simply set `html` to `false` and use the `text` property, as follows:

```
this.createTextField("theField_txt", 1, 0, 0, 200, 20);
theField_txt.html = false;
theField_txt.text = 'This field contains <B>HTML!</B>';
```

If the `embedFonts` and `html` properties are both `true` , every font face and variation used in `htmlText` must have corresponding font exported with the movie (as described under `embedFonts`). If we use Arial bold, Arial italic, and *A* bold italic in `htmlText` , then we must embed all three font variations. For example, suppose we create a text field calle `output_txt` . In the Properties inspector for our `output_txt` text field we select Arial, set to Italic. In the Charact Options dialog box, we embed All Characters of the Arial italic font. Then we set `output_txt` to display HTML. Fina we assign the following value to our text field:

```
output_txt.htmlText = '<P><I>My</I>, what <B>lovely</B>'
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

```
                + '<FONT SIZE="24">eyes</FONT> you have!</P>';
```

When the movie plays, the following text appears in the text field:

My

Everything else we assigned to `output` is missing! Only the italic text is rendered. The rest of the text requires other variations of the Arial font that we didn't embed—"what" and "eyes you have" are not italicized, and "lovely" is bolded.

Underlining is not considered a font variation, nor is font size or color. If, however, `embedFonts` is `false` , Flash rel entirely on the user's system for fonts, in which case normal, bold, and italic text will be rendered only if users have the appropriate font variant installed on their systems.

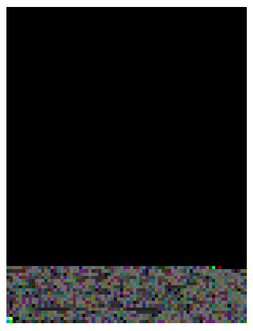
For further discussion of HTML usage in text fields, see Appendix E.

See Also

getURL( ) , *`TextField`* `.embedFonts` , *`TextField`* `.html` , *`TextField`* `.text` , the *TextFormat* class; Appendix E

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir



Chapter 18. ActionScript Language Reference

TextField.length Property Flash 6

the number of characters in the text fieldFlash 5; object-oriented version introduced in ***read-only***

`theField.length`

Description

The integer `length` property returns the number of characters in `theField.text` (similar to the `String.length` property), including whitespace. If `theField.htmlText` is assigned HTML source, `theField.length` still returns the number of characters in `theField.text` (which is the same as `htmlText`, but stripped of HTML tags). The `length` property of a text field is a character count, not a width; to access the pixel width of a field, use `theField._width`. To retrieve the value of a specific character in `theField`, use `theField.text.charAt(n)`, where `n` is the zero-relative character to retrieve.

Example

```
this.createTextField("theField_txt", 1, 0, 0, 200, 20);
theField_txt.text = "welcome";
trace(theField_txt.text.length);  // Displays: 7
```

See Also

String.charAt( ), `String.length`, `TextField.htmlText`, `TextField.text`, `TextField._width`

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Chapter 18. ActionScript Language Reference

TextField.maxChars Property — Flash 5

limits the allowed length of user inputFlash 6; accessible only via authoring tool in — ***read/write***

theField.maxChars

Description

Used with user-input text fields only, the integer `maxChars` property limits the amount of text a user can enter into a text field. It defaults to `null`, which allows an unlimited amount of text to be entered (as does the value `0`). Positive settings allow the specified number of characters to be entered; negative settings disable text entry completely. The `maxChars` property does not limit text changes made via ActionScript. The `text` and `htmlText` properties can be set to any value despite `maxChars` limitations.

The `maxChars` property is often used with input forms that require a certain format for their data. For example, we can use it to limit a date entry to a two-digit day, a two-digit month, and a four-digit year. To further curtail text entry, we can use the *`TextField`*`.restrict` property, which specifies a set of allowed input characters for a text field.

Example

The following code limits the number of characters that can be entered into `userName_txt` to 15:

```
this.createTextField("userName_txt", 1, 0, 0, 200, 20);
userName_txt.type = "input";
userName_txt.text = "your name here";
userName_txt.maxChars = 15;
```

See Also

`TextField``.htmlText`, *`TextField`*`.restrict`, *`TextField`*`.text`, *`TextField`*`._width`, *String.charAt( )*, *`String`*`.length`

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Chapter 18. ActionScript Language Reference

TextField.maxhscroll Property — Flash 6

the maximum distance text can be scrolled to the left — *read-only*

theField.maxhscroll

Description

The integer maxhscroll property indicates, in pixels, the maximum distance a field's text can be scrolled horizontally to the left. The maxhscroll property typically is used to detect when text has been moved left as far as it will go, or to force a text field to scroll its maximum distance.

When *theField*.type is set to "dynamic", maxhscroll tells us how far *theField*'s longest line overflows the right border (when hscroll is 0). When *theField*.type is set to "input", maxhscroll also includes a small buffer space at the right side of *theField*, (approximately one-fourth of the field's width). This buffer allows room for user input. When hscroll is equal to maxhscroll, *theField*'s longest line of text appears flush with the right border (minus the buffer space if *theField*.type is set to "input").

Figure 18-4 depicts maxhscroll for a dynamic text field (one with type set to "dynamic"). Text shown in gray is not displayed in the Flash Player but is shown for expository purposes.

Figure 18-4. The maxhscroll property for a dynamic text field

Figure 18-5 shows maxhscroll for the same field but with its type property set to "input". Note the increase in maxhscroll caused by the type change. The field in Table 18-2 has an hscroll of 150 so that the extra space allotted for user input is clearly visible.

Figure 18-5. The maxhscroll property for an input text field

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Example

The following code scrolls `title_txt` all the way to the right. For a more real-world example of `maxhscroll`, see Example 18-7 under *`TextField`*`.hscroll`.

```
this.createTextField("title_txt", 1, 0, 0, 30, 20);
title_txt.text = "Welcome to my web site.";
title_txt.hscroll = title_txt.maxhscroll;
```

See Also

`TextField``.bottomScroll`, *`TextField`*`.hscroll`, *`TextField`*`.maxscroll`, *`TextField`*`.scroll`; the *FScrollBar* component in Appendix G

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Chapter 18. ActionScript Language Reference

TextField.maxscroll Property — Flash 5

the last legal top line of a text fieldFlash 6; corresponds to the maxscroll property in Flash 4 and ***read-only***

theField.maxscroll

Description

The integer maxscroll property tells us the largest allowed scroll value for a text field. It represents the number of the last line in a text field that can be used as the top line in its viewable region. Note that maxscroll is not equal to the number of text lines; that value must be obtained manually, as discussed under *TextField*.bottomScroll.

See Example 18-7 under *TextField*.scroll for sample scrolling code.

See Also

TextField.bottomScroll, *TextField*.hscroll, *TextField*.maxhscroll, *TextField*.scroll; the *FScrollBar* component in Appendix G

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

TextField.multiline Property — Flash 5

Boolean; enables or disables multiline user input — ***read/write***
Flash 6; accessible only via authoring tool in

theField.multiline

Description

The Boolean `multiline` property dictates whether users can enter line breaks in *`theField`* using, say, the Return or Enter key. It applies to user input only and has no effect on text assigned with ActionScript. The default setting, `false`, prevents users from entering more than one line of text in the field, effectively disabling the Enter key during text entry (much like HTML's `<INPUT TYPE="TEXT">` tag). A setting of `true` allows users to enter more than one line of text in the text field (much like HTML's `<TEXTAREA>` tag).

To turn word-wrapping on for lines that exceed the field's viewable region, use *`TextField`*`.wordWrap`. To limit the amount of text a user can enter into a text field, use *`TextField`*`.maxChars`.

Example

The following code creates two text fields. The first field, `nameInput_txt`, allows a single line of input only; the second field, `addressInput_txt`, allows multiple lines of input:

```
// Create the nameInput_txt field
this.createTextField("nameInput_txt", 1, 0, 0, 200, 20);
nameInput_txt.type = "input";
nameInput_txt.border = true;
nameInput_txt.multiline = false;
// Create the addressInput_txt field
this.createTextField("addressInput_txt", 2, 0, 40, 200, 60);
addressInput_txt.type = "input";
addressInput_txt.border = true;
addressInput_txt.multiline = true;
```

See Also

TextField.maxChars, *TextField*.restrict, *TextField*.type, *TextField*.wordWrap

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Chapter 18. ActionScript Language Reference

TextField._name Property — Flash 6

the instance name of the TextField object — ***read/write***

theField._name

Description

The string _name property specifies a field's instance name, which is specified initially via the Property inspector at authoring time or via the *MovieClip.createTextField( )* method at runtime. Though rarely (if ever) necessary, setting _name reassigns the instance name for *theField*; thereafter, references to the old instance name yield undefined:

```
// Create a text field with an instance name of address_txt
this.createTextField("address_txt", 1, 0, 0, 200, 60);
// Reassign the instance name
address_txt._name = "location_txt";
// Now access the field through the new instance name
location_txt.text = "Toronto, Canada";
```

More commonly, the _name property is used to identify a specific text field when processing fields as a group. For example, the following code processes a hypothetical form. It finds the field email_txt on the current timeline (this) and then invokes a data-validating function on it:

```
for (var p in this) {
  if (this[p] instanceof TextField) {
    // Show each field's name
    trace("found " + this[p]._name);
    // Look for email_txt
    if (this[p]._name =  = "email_txt") {
      validateAddress(this[p].text);
    }
  }
}
```

See Also

MovieClip.createTextField( ), MovieClip._name; "The *for-in* Loop," in Chapter 8

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Chapter 18. ActionScript Language Reference

TextField.onChanged( ) Event Handler and Listener Event — Flash 6

invoked when user input is detected

```
theField.onChanged(changedField)
listener.onChanged(changedField)
```

Arguments

changedField

The *TextField* object for which input was detected.

Description

The *onChanged( )* event is triggered whenever the user enters or deletes text in `theField`. Typically, *onChanged( )* is used to trigger input-validation routines in response to the user's entries while the user is still typing. Possible uses include an inline spellchecker, an email validator, or an autocomplete feature for a text field.

Note that *onChanged( )* is not triggered when new text is assigned to `theField` through ActionScript. For example, this statement does not trigger *onChanged( )*:

```
theField_txt.text = "Some new text...";
```

If `theField.restrict` is `true` and the user attempts to enter a restricted character, then the keystroke is ignored and *onChanged( )* is not triggered.

The *onChanged( )* event can be handled with a callback function attached to the text field. The following code registers a callback function that responds to *onChanged( )*:

```
// Create a text field that accepts user input
this.createTextField("theField_txt", 1, 0, 0, 200, 20);
theField_txt.type = "input";
theField_txt.border = true;
// Assign the onChanged() callback using a function literal
theField_txt.onChanged = function (changedField) {
  trace("The user typed something in " + this);
};
```

Notice that from the body of the callback we refer to `theField_txt` using the `this` keyword; we could also have used the `changedField` parameter to obtain a reference to the text field that changed.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

The *onChanged( )* event can also be handled by an event listener that has been added using *addListener( )*, as specified by `listener`. When using an event listener, we must use the `changedField` parameter to determine which text field has changed. The following code registers an event listener that responds to *onChanged( )*:

```
changedListener = new Object();
changedListener.onChanged = function (changedField) {
  trace("The user typed something in " + changedField);
};
theField_txt.addListener(changedListener);
```

Using a callback function is simpler, but it allows only a single function to respond to the event. Any number of listener objects can respond to a single event. To stop a callback from handling the *onChanged( )* event, use the *delete* operator, as in:

```
// Make sure to omit the () operator after the function name!
delete theField_txt.onChanged;
```

To stop a listener from handling the *onChanged( )* event, use *removeListener( )*, as in:

```
theField_txt.removeListener(changedListener);
```

Bugs

When text is entered rapidly into a text field (for example, when many keys are pressed simultaneously), the new value of `theField.text` may not yet be set when the *onChanged( )* handler executes. When rapid typing is expected, use *setInterval( )* to wait until the new `text` is available.

See Also

TextField.addListener( ), *TextField.removeListener( )*, `TextField.type`; Chapter 10.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Chapter 18. ActionScript Language Reference

TextField.onKillFocus( ) Event Handler Flash 6

callback invoked when the field loses focus

```
theField.onKillFocus(newFocus)
```

Arguments

newFocus

The *TextField*, *MovieClip*, or *Button* object that now has focus, or `null` if no object received focus.

Description

The *onKillFocus( )* event is triggered when `theField` loses focus. A text field is said to have *focus* when it contains the text insertion point or when characters within it are selected. A field gains focus when:

- The user clicks it with the mouse or navigates to it with the Tab key.
- *Selection.setFocus( )* assigns focus programmatically.

A field loses focus (and *onKillFocus( )* is executed) when:

- The user clicks somewhere outside the field or navigates away from it via the Tab key, at which point focus may be transferred to whatever was tabbed to (a text field, movie clip, or button). Clicking a button or movie clip does not give the button or clip focus. If focus was transferred, the newly focused object is accessible within *onKillFocus( )* via the `newFocus` parameter; otherwise, `newFocus` is `null`.
- *Selection.setFocus( )* assigns focus to another object programmatically. The newly focused object is accessible within *onKillFocus( )* via `newFocus`.

When a text field's `selectable` property is `false`, the field cannot be focused by the user but can be focused programmatically with *Selection.setFocus( )*.

The following code creates a callback function that responds to *onKillFocus( )*. Run it in Test Movie mode and try clicking on and off the text field. The Output window displays a message when the event is triggered.

```
// Create a text field that accepts user input
this.createTextField("theField_txt", 1, 0, 0, 200, 20);
theField_txt.type = "input";
theField_txt.border = true;
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

```
// Assign the onKillFocus() callback using a function literal
theField_txt.onKillFocus = function (newFocus) {
  trace(this + " lost focus. Focus is now " + newFocus);
};
```

Notice that from the body of the callback we refer to `theField_txt` using the `this` keyword. To stop a callback from handling the *onKillFocus( )* event, use the *delete* operator, as in:

```
// Make sure to omit the () operator after the function name!
delete theField_txt.onKillFocus;
```

The *onKillFocus( )* event can be used to trigger the validation of text field input when the user has finished entering text. Or, in combination with *onSetFocus( )*, it can bring attention to a focused text field with color (see the Example under *TextField.onSetFocus( )*). To capture all focus events centrally, rather than for a single instance, use *Selection.onSetFocus( )*.

See Also

Selection.onSetFocus( ), *Selection.setFocus( )*, *TextField.onSetFocus( )*, *`TextField`*`.selectable`, *`TextField`*`.tabEnabled`, *`TextField`*`.tabIndex`, *`TextField`*`.type`; Chapter 10

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Chapter 18. ActionScript Language Reference

TextField.onScroller( ) Event Handler and Listener Event — Flash 6

invoked when any scroll property changes

```
theField.onScroller(scrolledField)
listener.onScroller(scrolledField)
```

Arguments

scrolledField

The *TextField* object that changed scroll position.

Description

The *onScroller( )* event is triggered whenever any of the following text field properties changes: `scroll`, `maxscroll`, `hscroll`, or `maxhscroll`. It tells us that *`theField`* has been scrolled vertically or horizontally, or that *`theField`* has new content that changes its maximum vertical or horizontal scrolling range. Typically, *onScroller( )* is used to synch a scrollbar interface with the content of *`theField`*. For example, if the user enters more text than will fit in a single-line text field, the text will scroll horizontally and *onScroller( )* will fire, allowing code to update the width and position of a horizontal scrollbar.

The *onScroller( )* event can be handled with a callback function attached to the text field. The following code registers a callback function that responds to *onScroller( )*. Run it in Test Movie mode and enter multiple lines of text into the field to see it work.

```
// Create a text field that accepts multiple lines of user input
this.createTextField("theField_txt", 1, 0, 0, 200, 20);
theField_txt.type = "input";
theField_txt.border = true;
theField_txt.multiline = true;
// Assign the onScroller() callback using a function literal
theField_txt.onScroller = function (scrolledField) {
  trace("scroll is "     + this.scroll);
  trace("maxscroll is "  + this.maxscroll);
  trace("hscroll is "    + this.hscroll);
  trace("maxhscroll is " + this.maxhscroll);
};
```

Notice that from the body of the callback we refer to `theField_txt` using the `this` keyword; we could also have used the *`scrolledField`* parameter.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

The *onScroller( )* event can also be handled by an event listener that has been added using *addListener( )*, as specified by `listener`. When using an event listener we must use the `scrolledField` parameter to determine which text field was scrolled. The following code registers an event listener that responds to *onScroller( )*:

```
scrollerListener = new Object();
scrollerListener.onScroller = function (scrolledField) {
  trace("scroll is "     + scrolledField.scroll);
  trace("maxscroll is "  + scrolledField.maxscroll);
  trace("hscroll is "    + scrolledField.hscroll);
  trace("maxhscroll is " + scrolledField.maxhscroll);
};
theField_txt.addListener(scrollerListener);
```

Using a callback function is simpler, but it allows only a single function to respond to the event. Any number of listener objects can respond to a single event. To stop a callback from handling the *onScroller( )* event, use the *delete* operator, as in:

```
// Make sure to omit the () operator after the function name!
delete theField_txt.onScroller;
```

To stop a listener from handling the *onScroller( )* event, use *removeListener( )*, as in:

```
theField_txt.removeListener(scrollerListener);
```

Usage

Note that simply clicking in a text field causes *onScroller( )* to fire.

Bugs

In Flash 6, when a text field's `scroll` or `hscroll` properties are set on the same frame as the creation of the field, *onScroller( )* is not triggered. Even when those properties are set on the frame after the creation of the field, *onScroller( )* may not execute:

```
// Create a text field.
this.createTextField("theField_txt", 1, 0, 0, 200, 40);
theField_txt.text = "a\nb\nc\nd\ne\nf\ng\nh\ni\nj\nk\nl\nn\nm\no\np";
// Assign the onScroller() callback.
theField_txt.onScroller = function (scrolledField) {
  trace("scroll is " + this.scroll);
  trace("maxscroll is " + this.maxscroll);
};
// Scroll the field. onScroller() will not execute.
theField_txt.scroll = 4;
```

To force the *onScroller( )* event handler to execute, invoke it manually, as in:

```
theField_txt.onScroller();
```

Unfortunately, there is no easy way to force all *onScroller( )* listeners to execute manually. (You'd have to keep a list of all listeners and execute the *onScroller( )* method of each individually.)

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Example

The following code detects when `theField_txt` has been scrolled all the way to the bottom:

```
theField_txt.onScroller = function (scrolledField) {
  if (this.scroll =  = this.maxScroll) {
    trace("End of field reached.");
  }
}
```

See Also

TextField.addListener( ), `TextField.hscroll`, `TextField.maxhscroll`, `TextField.maxscroll`, *TextField.removeListener( )*, `TextField.scroll`; Chapter 10; the *FScrollBar* component in Appendix G

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Chapter 18. ActionScript Language Reference

TextField.onSetFocus( ) Event Handler Flash 6

callback invoked when the field gains focus

```
theField.onSetFocus(oldFocus)
```

Arguments

oldFocus

The *TextField*, *MovieClip*, or *Button* object that previously had focus, or `null` if no object was previously focused.

Description

The *onSetFocus( )* event is triggered when `theField` gains focus. A text field is said to have *focus* when it contains the text insertion point or when characters within it are selected. A field gains focus when:

- The user clicks it with the mouse or navigates to it with the Tab key.
- *Selection.setFocus( )* assigns focus programmatically.

When a text field's `selectable` property is `false`, the field cannot be focused by the user, but it can be focused programmatically with *Selection.setFocus( )*.

The following code registers a callback function that responds to *onSetFocus( )*:

```
// Create a text field that accepts user input
this.createTextField("theField_txt", 1, 0, 0, 200, 20);
theField_txt.type = "input";
theField_txt.border = true;
// Assign the onSetFocus() callback using a function literal
theField_txt.onSetFocus = function (newFocus) {
  trace(this + " now has focus. Old focus was " + newFocus);
};
```

Notice that from the body of the callback we can refer to `theField_txt` using the `this` keyword. To stop a callback from handling the *onSetFocus( )* event, use the *delete* operator, as in:

```
// Make sure to omit the () operator after the function name!
delete theField_txt.onSetFocus;
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

To capture all focus events centrally, rather than for a single instance, use *Selection.onSetFocus( )*.

Example

The following code uses *onSetFocus( )* and *onKillFocus( )* to highlight a field with color when it is focused, providing a nice interface cue for the user:

```
this.createTextField("userName_txt", 1, 0, 0, 200, 20);
userName_txt.type = "input";
userName_txt.border = true;
userName_txt.background = true;
userName_txt.backgroundColor = 0xA4660B;
userName_txt.onSetFocus = function (oldFocus) {
  this.backgroundColor = 0xF5BF70;
}
userName_txt.onKillFocus = function (newFocus) {
  this.backgroundColor = 0xA4660B;
}
```

To apply this effect to all text fields in a movie, use:

```
TextField.prototype.onSetFocus = function (oldFocus) {
  this.backgroundColor = 0xF5BF70;
}
TextField.prototype.onKillFocus = function (newFocus) {
  this.backgroundColor = 0xA4660B;
}
```

See Also

Selection.onSetFocus( ), *Selection.setFocus( )*, *TextField*.background, *TextField*.backgroundColor, *TextField.onKillFocus( )*, *TextField*.selectable, *TextField*.tabEnabled, *TextField*.tabIndex, *TextField*.type; Chapter 10

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Chapter 18. ActionScript Language Reference

TextField._parent Property — Flash 6read/write (write has no meaningful effect)

a reference to the movie clip in which the text field resides

theField._parent

Description

The _parent property is an object reference to the movie clip that contains *theField*. For example, if we have a text field named title_txt in a clip named mainSection_mc, then title_txt._parent refers to mainSection_mc. Compared to its cousin, *MovieClip*._parent, *TextField*._parent is rarely used, but it may occasionally be handy when working with text fields grouped in separate movie clips.

Usage

Though it is possible to reassign the _parent property to another clip (or any other value), doing so has little purpose—only the reference is changed, not the physical hierarchy of the clip and text field.

See Also

MovieClip._parent, *TextField*._name

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

TextField.password Property — Flash 5

obscure characters displayed in the fieldFlash 6; accessible only via authoring tool in — *read/write*

theField.password

Description

The Boolean password property specifies whether a field's characters should be displayed normally (false) or obscured on screen for privacy (true). When true, all characters, including spaces, are displayed as asterisks (*). For example, the words "hi there" are displayed as "********". This allows users to enter text without casual onlookers seeing it.

Any text entered while password is true is not altered and can be retrieved as usual through the text property. The password property does not affect other line display properties such as multiline and wordWrap; text in a password-enabled field can both wrap and span multiple lines. The Cut and Copy commands do not function in a password text field.

> Fields for which password is true are not secure despite being masked on screen. Their content is still visible if, say, it is transmitted to a server-side script. For information on securing a Flash application, see http://www.macromedia.com/desdev/security and http://www.macromedia.com/desdev/mx/flash/extreme/extreme003.html.

Example

The following code creates typical username and password text input fields:

```
this.createTextField("nameInput_txt", 1, 0, 0, 200, 20);
nameInput_txt.type = "input";
nameInput_txt.border = true;
this.createTextField("passInput_txt", 2, 0, 30, 200, 20);
passInput_txt.type = "input";
passInput_txt.border = true;
passInput_txt.password = true;
```

See Also

TextField.multiline, *TextField*.text, *TextField*.type, *TextField*.wordWrap

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

TextField.removeListener( ) Method Flash 6

cancels event notices for the specified listener

```
theField.removeListener(listener)
```

Arguments

listener

An object that was previously registered as a listener for `theField`.

Returns

A Boolean: `true` if `listener` was found and removed, `false` if `listener` was not registered as an event listener for `theField`. The return value is useful for debugging code that removes listeners.

Description

The *removeListener( )* method stops `listener` from receiving notification of *onScroller( )* and *onChanged( )* events occurring on `theField`. It should be used only after `listener` has been registered as an event listener via *addListener( )*. The `listener` object is not deleted, but it can be removed and added repeatedly. Event listeners normally are removed when the task they perform is no longer required. For example, a listener that controls a scrollbar could be removed when an application enters a disabled state due to, say, the launching of a modal dialog box.

Example

```
// Make a text field that accepts user input.
this.createTextField("theField_txt", 1, 0, 0, 200, 20);
theField_txt.border = true;
theField_txt.type = "input";
// Create a listener.
txtListener = new Object();
// Add the onScroller function to the listener.
txtListener.onScroller = function (tfObject) {
  trace(tfObject + " scroll property changed.");
}
// This adds the listener to theField_txt.
theField_txt.addListener(txtListener);
// This removes the listener from theField_txt.
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

```
theField_txt.removeListener(txtListener);
// This also removes the listener from theField_txt, but it
// displays an error message if the listener is not found.
if (!theField_txt.removeListener(txtListener)) {
  trace("ERROR. No such listener on theField_txt.");
}
```

See Also

TextField.addListener( ), *TextField.onChanged( )*, *TextField.onScroller( )*; Chapter 10

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Chapter 18. ActionScript Language Reference

TextField.removeTextField( ) Method Flash 6

deletes a runtime-created text field

```
theField.removeTextField()
```

Description

When `theField` is a text field created at runtime with *MovieClip.createTextField( )*, the *removeTextField( )* method deletes `theField` and removes its associated visual representation from the screen. Subsequent references to `theField` or any of its properties or methods yield `undefined`. When `theField` is a text field created at authoring time with the Text tool, *removeTextField( )* has no effect. To remove an authoring time text field, place it in a movie clip and remove the entire clip with *MovieClip.removeMovieClip( )*.

Example

```
// Make a text field
this.createTextField("theField_txt", 1, 0, 0, 200, 20);
// Then sometime later...delete the text field
theField_txt.removeTextField();
```

See Also

MovieClip.createTextField( ), *MovieClip.removeMovieClip( )*

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

TextField.replaceSel( ) Method Flash 6

replace selected text (without disturbing formatting)

```
theField.replaceSel(newText)
```

Arguments

newText

The string of new characters that will replace the current selection.

Description

The *replaceSel( )* method deletes the selected characters in `theField` and replaces them with `newText` . If no characters are selected in `theField` , *replaceSel( )* simply inserts `newText` at `theField` 's insertion point, or before the first character if `theField` is not focused. The new amalgamated value is reflected by `theField.text` .

While *replaceSel( )* 's use may seem limited, it is actually one of the most important *TextField* methods because it provides the only means of inserting new text into a text field without disturbing existing character formatting. To update a text field's value using *replaceSel( )* , we must first focus the field, then position the insertion point, and then insert the text, as follows:

```
// Focus the field
Selection.setFocus("theField_txt");
// Position insertion point after the fourth character
Selection.setSelection(4, 4);
// Insert new text
theField_txt.replaceSel("text to insert");
```

Obviously, this process is tedious. See Example 18-6 for a custom method that takes care of the selection work.

To apply a new format to text inserted by *replaceSel( )* , we must invoke *setNewTextFormat( )* after the insertion point is set with *Selection.setSelection( )* . See the Usage heading under *TextField.setNewTextFormat( )* for details.

In some cases, we want to replace a selection made by the user when a button is clicked (for example, when inserting text in a word processing-style application). This proves to be trickier than expected because the selection is lost when focus changes to the button during the click. To prevent the selection from being lost, we record it in the button's rollover event handler, as follows:

```
replace_btn.onRollover = function () {
  // Record which field was focused
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

```
  oldFocus = Selection.getFocus();
  // Record the selection span
  oldBegin = Selection.getBeginIndex();
  oldEnd = Selection.getEndIndex();
}
replace_btn.onRelease = function () {
  // If a text field was the last object focused...
  if (eval(oldFocus) instanceof TextField) {
    // ...focus it
    Selection.setFocus(oldFocus);
    // Then set the selection and insert the new text
    Selection.setSelection(oldBegin, oldEnd);
    eval(oldFocus).replaceSel("text to insert");
  }
}
```

Example

Example 18-6 adds an *appendText( )* method to all *TextField* objects. The method inserts new text at the end of the field, using *replaceSel( )* .

Example 18-6. Custom appendText( ) function using replaceSel( )

```
/*
 * Method: TextField.appendText
 *   Desc: A convenient method for appending text to a
 *         text field without disturbing its formatting.
 * Params: theText     The text to append.
 */
TextField.prototype.appendText = function (theText) {
  // Remember the original scroll because replaceSel( ) can change scroll.
  var origScroll = this.scroll;
  // Focus the field.
  Selection.setFocus(this);
  // Position insertion point after the last character.
  Selection.setSelection(this.length, this.length);
  // Append the text to the field.
  this.replaceSel(theText);
  // Reset the scroll to where it was before we added any text.
  this.scroll = origScroll;
}
```

Reader Exercise: Adapt the *appendText( )* handler from Example 18-6, or use the example from this entry's Description, to create a custom *replaceText( )* function that accepts starting and ending character indexes to specify the character span to be replaced.

See Also

Selection.setFocus( ) , *Selection.setSelection ( )* , *TextField.setNewTextFormat( )* , `TextField .text`

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

TextField.restrict Property — Flash 6

limit text entry to a series of characters — *read/write*

```
theField.restrict = restrictedChars;
```

Arguments

restrictedChars

Although not strictly an argument, `restrictedChars` is a string containing the allowable characters for the field.

Description

The string `restrict` property specifies the characters that the user can enter into `theField`. When restrict contains `null` (the default) or `undefined`, any character can be entered. When `restrict` contains a nonempty string of characters, only those characters are allowed as user input; attempts to enter other characters will simply be ignored. The following example limits text entry to the letters *y* and *n*. It also uses `maxChars` to limit text entry length to a single character:

```
this.createTextField("yesnoInput_txt", 1, 0, 0, 200, 20);
yesnoInput_txt.type = "input";
yesnoInput_txt.border = true;
yesnoInput_txt.restrict = "yn";
yesnoInput_txt.maxChars = 1;
```

Note that `restrict` operates according to character code points, not keyboard keys, so upper- and lowercase letters, such as *Y* and *y*, must be listed separately, as in:

```
yesnoInput_txt.restrict = "ynYN";
```

Interestingly, when only one case is specified, text entry is converted to that case. For example, to force the letters *y* and *n* to be converted to *Y* and *N*, use:

```
yesnoInput_txt.restrict = "YN";
```

Characters can be specified with string literals, as we have just seen, or with Unicode escape sequences in the form `\uxxxx`, where `xxxx` is the hexadecimal code point of the allowed character. Unicode notation normally is used with international characters. The following code assigns "ynYN" using escape sequences:

```
yesnoInput_txt.restrict = "\u0079\u006e\u0059\u004e";
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Specifying many characters for inclusion in a text field can be tedious, so `restrict` also supports special sequences that represent character ranges and exclusions. These will be familiar to programmers accustomed to regular expressions. The "-" character marks a character range. For example:

```
// Limit ageInput_txt to the numbers: 0123456789
ageInput_txt.restrict = "0-9";
```

Multiple ranges are listed without spaces, as in:

```
// Limit passInput_txt to letters (upper- and lowercase) and numbers only
passInput_txt.restrict = "a-zA-Z0-9";
```

Characters included in a range are those between the Unicode code points of the specified first and last characters, inclusive. For example, the following limits text entry to common punctuation symbols (hex code points 21 to 2F):

```
theField_txt.restrict = "\u0021-\u002F";
```

Normally, characters assigned to `restrict` are those *included* in the allowable set of input. The ^ character lets us specify characters to be *excluded* from allowable input. Any character following the ^ character cannot be entered into *`theField`*. For example:

```
// Include all characters except the numbers
theField_txt.restrict = "^0-9";
// Include upper- and lowercase letters, but excluding x and X
theField_txt.restrict = "a-zA-Z^xX";
```

To include the \, ^, or - characters, use \\\\, \\^, and \\-. For example, the following code restricts text entry to the letters a-z and the characters \, ^, and -:

```
theField_txt.restrict = "a-z\\^\\-\\\\";
```

The `restrict` property does not limit text changes made by a script; it applies to user input only. That is, the `text` and `htmlText` properties can be set to any value, regardless of `restrict`'s limitations. Also, `restrict` does not dictate which characters are exported as font outlines with the movie; it merely limits text entry. Font outline export can be set via the authoring tool only, using the Character button in the text field Property inspector. See *`TextField`*`.embedFonts` for details on exporting fonts.

Bugs

Setting `restrict` to the empty string (`""`) should entirely disable text input; however, in Flash Player 6, it allows any character to be entered, exactly as if `restrict` were set to `null`. To disable input entirely, set the field's `type` property to "dynamic" and its `selectable` property to `false`.

See Also

`TextField``.embedFonts`, *`TextField`*`.maxChars`, *`TextField`*`.password`, *`TextField`*`.selectable`, *`TextField`*`.type`; "Unicode-style escape sequences," in Chapter 4; Appendix B

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Chapter 18. ActionScript Language Reference

TextField._rotation Property — Flash 6

rotation, in degrees, of a text field — ***read/write***

theField._rotation

Description

The floating-point _rotation property specifies the number of degrees *theField* is rotated about its parent movie registration point. Both author-time and programmatic adjustments are reflected in _rotation. Numbers in the range to 180.0 rotate *theField* clockwise. Numbers in the range of 0 to -180.0 rotate it counterclockwise. See *MovieCl* ._rotation for details on various positive and negative values for rotation, including out-of-range values.

Usage

In order for a text field to be visible when it is rotated, its font *must* be embedded! See *TextField* .embedFonts.

Example

The following code spins a text field in a circle. It assumes that the Arial font is available on the system at export time.

```
this.createTextField("spinner_txt", 1, 200, 200, 100, 20);
spinner_txt.embedFonts = true;
spinner_txt.text = "moock.org";
spinner_txt.setTextFormat(new TextFormat("Arial"));
spinner.rotInterval = setInterval(function () {spinner_txt._rotation+=10}, 1
```

See Also

MovieClip ._rotation, *TextField* .embedFonts

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Chapter 18. ActionScript Language Reference

TextField.scroll Property — Flash 5

the current top line displayed in a text fieldFlash 6; corresponds to the scroll property in Flash 4 and ***read/write***

theField.scroll

Description

The integer scroll property represents the number (index) of the line currently displayed as the first line in a field's viewable region. When the text in a text field spans more lines than can be accommodated by the physical viewable regior of the field, extra lines of text are hidden but are still part of the text field. The top line is number 1, and line numbers increase for every line in the text field, including those that exceed the viewable boundaries of the field, as shown in Figure 18-6.

Figure 18-6. Text field line indexes

Setting the value of scroll scrolls the text field, making the supplied line number the top line in the field's viewable region. For example, if we set the scroll property of the text field shown in Figure 18-6 to 3, the text field displays:

```
or
thoughts
in
```

Example

In combination, the scroll and maxscroll properties can be used to create scroll buttons for a text field. This example scrolls text down one line for each click of a button:

```
scrollButton.onPress = function () {
  if (theField_txt.scroll < theField_txt.maxscroll) {
    theField_txt.scroll++;
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

```
  }
};
```

And here's how we scroll text up one line with each click:

```
scrollButton.onPress = function () {
  if (theField_txt.scroll > 1) {
    theField_txt.scroll--;
  }
};
```

For an example of simple scroll buttons used in a movie, download the sample scrollers posted at the online Code Depot. As of Flash 6, the FScrollBar Flash UI Component can service most scrolling needs, so custom-developed scrolling widgets rarely are necessary. However, `scroll` has other interesting applications, such as autoscrolling, shown in Example 18-7 (also available at the Code Depot). Example 18-7 adds to the *TextField* class scrolling methods that scroll up, scroll down, and scroll automatically according to the specified number of lines and speed. When the autoscroller reaches the top or bottom of the text field, it invokes the field's custom *onTopReached( )* or *onBottomReached( )* callback allowing the code to respond by, say, showing the next page in a slideshow.

Example 18-7. A TextField autoscroller

```
// Make a text field
this.createTextField("theField_txt", 1, 0, 0, 200, 60);
theField_txt.border = true;
theField_txt.text = "line1\nline2\nline3\nline4\nline5\nline6\nline7\nline8"
// Add a method to scroll any text field down
TextField.prototype.scrollDown = function (numLines) {
  if (this.scroll + numLines < this.maxscroll) {
    this.scroll += numLines;
  } else {
    this.scroll = this.maxscroll;
    this.stopAutoScroll();
    this.onBottomReached();
  }
}
// Add a method to scroll any text up
TextField.prototype.scrollUp = function (numLines) {
  if (this.scroll - numLines > 1) {
    this.scroll -= numLines;
  } else {
    this.scroll = 1;
    this.stopAutoScroll();
    this.onTopReached();
  }
}
// Add a method to autoscroll any text field
// Scrolls every speed milliseconds by numLines lines, either up or down
TextField.prototype.startAutoScroll = function (speed, numLines, dir) {
  if (typeof speed =  = "number" && typeof numLines =  = "number") {
    if (dir =  = "down") {
      this.autoScroller = setInterval(this, "scrollDown", speed, numLines);
    } else if (dir =  = "up") {
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

```
      this.autoScroller = setInterval(this, "scrollUp", speed, numLines);
    }
  }
}
// Add a method to stop autoscrolling
TextField.prototype.stopAutoScroll = function () {
  clearInterval(this.autoScroller);
}
// When we reach the bottom, start scrolling back up, quickly
theField_txt.onBottomReached = function () {
  this.startAutoScroll(100, 1, "up");
}
// When we reach the top, start scrolling back down
theField_txt.onTopReached = function () {
  this.startAutoScroll(1000, 1, "down");
}
// Scroll by 1 line every second
theField_txt.startAutoScroll(1000, 1, "down");
// To stop scrolling, use:
// theField_txt.stopAutoScroll();
```

See Also

TextField .bottomScroll, *TextField* .hscroll, *TextField* .maxhscroll, *TextField* .scroll; the *FScrollBar* component in Appendix G

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Chapter 18. ActionScript Language Reference

TextField.selectable Property — Flash 6

Boolean; enables or disables selection by the user ***read/write***

theField.selectable

Description

The Boolean selectable property governs whether the characters in *theField* can be selected by the user. By default, all text fields can be selected (selectable is true). To create a nonselectable text field for display only, set its selectable property to false:

```
this.createTextField("output_txt", 1, 0, 0, 200, 20);
output_txt.selectable = false;
output_txt.text = "This is output only.";
```

To create a text field that can receive user input, set its type property to "input" and ensure that selectable is true:

```
this.createTextField("input_txt", 1, 0, 0, 200, 20);
input_txt.selectable = true;
input_txt.border = true;
input_txt.type = "input";
input_txt.text = "This field can accept input.";
```

When selectable is false, a field cannot receive user input, nor can it be focused by the user. However, a text field can always be focused and selected programmatically with the *Selection* object, regardless of the value of its selectable property.

See Also

The *Selection* object, *TextField*.maxChars, *TextField.onKillFocus( )*, *TextField.onSetFocus( )*, *TextField*.restrict, *TextField*.type

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

TextField.setNewTextFormat( ) Method Flash 6

sets the default formatting for the text field

theField.setNewTextFormat(*textFormatObject*)

Arguments

textFormatObject

An instance of the *TextFormat* class.

Description

The *setNewTextFormat( )* method sets the default formatting for *theField* . After it is invoked, any text added to *th* via *TextField .text* or *TextField.replaceSel( )* , or by the user, is formatted according to the property values of *textFormatObject* . However, *setNewTextFormat( )* has no effect on characters already in *theField* . To refo characters already in *theField* , use *setTextFormat( )* .

The *setNewTextFormat( )* method is perfect for establishing the default formatting of a text field whose content is likely t For example, suppose we want to display a player's score in blue, 10-point Courier New. Each time the player earns point score changes and we update our score text field. To ensure that the formatting of the text field is retained for each update *setNewTextFormat( )* immediately after the field is created:

```
// Create the text field.
this.createTextField("score_txt", 1, 50, 50, 100, 20);
// Assign the default TextFormat. Make sure this comes before any text is as
score_txt.setNewTextFormat(new TextFormat("Courier New", 10, 0x0000FF));
// Set score output.
score_txt.text = "0000";
// Change score output. Formatting is retained.
score_txt.text = "0020";
```

To examine the existing default format for *theField* , use *getNewTextFormat( )* . For a detailed description of *TextFc* objects, see the *TextFormat* class. Note that text can also be formatted with HTML, as described under *TextField .h* *TextField .htmlText* .

Usage

When the insertion point is placed in a text field by the user or with *Selection.setSelection( )* , the field's default text form match the character to the insertion point's right (or left, in the case of the last character). To prevent any subsequent user i from picking up the formatting of existing text, we must manually restore the default text format with an interval, as show

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Example 18-8. The workaround, however, is not perfect—on slower machines, the interval cannot fire fast enough to inter every keystroke and the occasional character takes on the formatting of existing text.

Example 18-8. Resetting new text format after selection

```
// Create the text field and set it to accept user input
this.createTextField("theField_txt", 1, 0, 0, 150, 40);
theField_txt.type = "input";
// Store the original new text format in a property
theField_txt.newTextFormat = theField_txt.getNewTextFormat();
// Define a callback to handle setFocus events
theField_txt.onSetFocus = function (oldFocus, newFocus) {
  // If the field is focused, start an interval that
  // continuously resets the original newTextFormat
  if (newFocus =  = this) {
    this.restoreInterval = setInterval(this, "restoreNewTextFormat", 1);
  } else if (oldFocus =  = this) {
    // Stop resetting if the field is unfocused
    clearInterval(this.restoreInterval);
  }
}
// Create the interval function to reset the newTextFormat
theField_txt.restoreNewTextFormat = function () {
  this.setNewTextFormat(this.newTextFormat);
}
// Register the text field to receive setFocus events
Selection.addListener(theField_txt);
// Assign the field some text for display
theField_txt.text = "What time is it?"
// Create a TextFormat object with a font property of "Arial"
sansFormat = new TextFormat("Arial");
// Apply the format to the word 'What', characters 0 to 3 (inclusive)
theField_txt.setTextFormat(0, 4, sansFormat);
```

When text is inserted with the *replaceSel( )* method, it's easy to force the appropriate new text format—simply invoke *setNewTextFormat( )* after the call to *Selection.setSelection( )* , as shown in Example 18-9.

Example 18-9. Setting the new text format before replacing a selection

```
// Focus the field we're going to update
Selection.setFocus(theField_txt);
// Position the insertion point at the first character
Selection.setSelection(1,1);
// *Now* set the newTextFormat
theField_txt.setNewTextFormat(theField_txt.newTextFormat);
// Proceed with replaceSel() call
theField_txt.replaceSel("this is the new text");
```

See Also

The *Selection* class, *TextField.getTextFormat( )* , *TextField.getNewTextFormat( )* , *TextField.replaceSel ( )* ,

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

TextField.setTextFormat( ) , the *TextFormat* class

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

TextField.setTextFormat( ) Method Flash 6

sets the formatting for a specified range of characters

```
theField.setTextFormat(textFormatObject)
theField.setTextFormat(beginIndex, textFormatObject)
theField.setTextFormat(beginIndex, endIndex, textFormatObject)
```

Arguments

textFormatObject

An instance of the *TextFormat* class.

beginIndex

The index of the first character whose formatting will be set by `textFormatObject`. Must be a nonnegative integer.

endIndex

The index following the last character whose formatting will be set by `textFormatObject`. Must be a nonnegative integer.

Description

The *setTextFormat( )* method formats the span of characters from `beginIndex` to `endIndex` according to the properties of `textFormatObject`. Any `null` or `undefined` properties of `textFormatObject` have no effect on the target characters. The *setTextFormat( )* method is invoked with one, two, or three arguments. When *setTextFormat( )* is invoked with two arguments, the format applies to the single character at `beginIndex`. For example, here we apply a bold format to the first character of a text field:

```
// Create the text field
this.createTextField("theField_txt", 1, 0, 0, 150, 40);
// Assign the field some text for display
theField_txt.text = "Products and Services";
// Create a TextFormat object that specifies bold text
boldFormat = new TextFormat();
boldFormat.bold = true;
// Apply the format to the letter 'P', character 0
theField_txt.setTextFormat(0, boldFormat);
```

When *setTextFormat( )* is invoked with three arguments, the format applies to the span of characters from `beginIndex` to `endIndex` - `1`. For example, this code italicizes the word "and":

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

```
// Create a TextFormat object that specifies italic text
italicFormat = new TextFormat();
italicFormat.italic = true;
// Apply the format to the word 'and', characters 9 to 11 (inclusive)
theField_txt.setTextFormat(9, 12, italicFormat);
```

When *setTextFormat( )* is invoked with one argument, the format applies to the all the characters in the field. For example, the following sets the font for the entire text of `theField_txt` to Arial:

```
// Create a TextFormat object that specifies the font "Arial"
arialFormat = new TextFormat();
arialFormat.font = "Arial";
// Apply the format to the entire text
theField_txt.setTextFormat(arialFormat);
```

Notice that the sequential formatting is not destructive; formatting changes are applied only for properties that are set, so the letter "P" remains bolded and the word "and" remains italicized.

Setting the value of a field's `text` property destroys all formatting for the field. For example, if we now add the word "Overview" to `theField_txt.text`, all characters are rerendered in the new text format for the field:

```
// Resets theField_txt's formatting
theField_txt.text += "Overview";
```

To add content to a text field without affecting its formatting, use the *replaceSel( )* method.

> Don't confuse *setNewTextFormat( )* with *setTextFormat( )*; the former sets the format used for text that will be added to *`theField`* in the future, while the latter sets the format for characters already in *`theField`*. Both methods are valid, but the order in which they are used is crucial. Use *setTextFormat( ) after* setting the `text` property; use *setNewTextFormat( ) before* setting the `text` property.

Here, using *setTextFormat( )*, we first assign our text and then apply formatting to it:

```
// Create the text field
this.createTextField("title_txt", 1, 0, 0, 150, 40);
// Assign the field some text for display
title_txt.text = "What's New?";
// Now create and apply our TextFormat object
title_txt.setTextFormat(new TextFormat("Arial", 14, 0xFF0000));
```

If we mistakenly apply our format before the text is assigned, nothing happens:

```
// Oops! Wrong order...
title_txt.setTextFormat(new TextFormat("Arial", 14, 0xFF0000));
title_txt.text = "What's New?";
```

Conversely, when using *setNewTextFormat( )*, we must be sure to apply our format before assigning text to the text field:

```
title_txt.setNewTextFormat(new TextFormat("Arial", 14, 0xFF0000));
title_txt.text = "What's New?";
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

The *setNewTextFormat( )* method is convenient when text is likely to change, because it applies to all future assignments for the field:

```
// The text "Latest News" is still red, 14-point Arial.
title_txt.text = "Latest News";
```

For a detailed description of *TextFormat* objects, see the *TextFormat* class. Note that text can also be formatted with HTML, as described under *TextField*.html and *TextField*.htmlText.

Example

The following code adds to all *TextField* objects a *rainbow( )* method that randomly colors a field's text:

```
// Here's the method
TextField.prototype.rainbow = function () {
  for (var i = 0; i < this.length; i++) {
    var randColor = Math.floor(Math.random() * 0xFFFFFF);
    this.setTextFormat(i, new TextFormat(null, null, randColor));
  }
}
// Now let's create a text field to try it on
this.createTextField("theField_txt", 1, 50, 50, 160, 20);
// Set some text
theField_txt.text = "Ooh, what pretty colors!";
// Try it out!
theField_txt.rainbow();
```

See Also

TextField.getNewTextFormat( ), *TextField.getTextFormat( )*, *TextField*.html, *TextField*.htmlText, *TextField.replaceSel( )*, *TextField.setNewTextFormat( )*, the *TextFormat* class

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Chapter 18. ActionScript Language Reference

TextField.tabEnabled Property — Flash 6

Boolean; includes or excludes the field from the current tab order — ***read/write***

theField.tabEnabled

Description

The Boolean `tabEnabled` property governs whether *`theField`* is part of the *tab order* for the text fields, movie clips, and buttons currently on screen. The tab order is the sequence in which objects are focused when the Tab key is pressed in a movie. Using the Tab key during text entry, a user can move between text fields in a form.

However, there are two distinct tab orders in a Flash movie: the *automatic tab order* and the *custom tab order.* The automatic tab order is Flash's default tab order, used when no custom tab order is defined. It includes all buttons and input text fields currently on screen and orders them from left to right, then top to bottom. Text fields with `type` set to "dynamic" are not included. The custom tab order is an ActionScript-defined tab order including an arbitrary sequence of *Button*, *MovieClip*, and *TextField* objects. To define a custom tab order, we set the `tabIndex` property for each button, movie clip, or text field we want included in the order. When at least one object currently on screen has a `tabIndex` property defined, the custom tab order is used.

For text fields, the `tabEnabled` property has two applications:

- It can exclude an input text field with a defined `tabIndex` from the custom tab order.
- It can exclude an input text field from the automatic tab order.

Neither scenario is particularly common because input text fields normally should be keyboard-accessible. To disable an input text field temporarily, we can simply set its `type` to "dynamic", which also removes it from the tab order.

Note that the behavior of `tabEnabled` depends on the value of the `tabIndex` property. For reference, Table 18-24 shows the results of the possible `tabEnabled` and `tabIndex` combinations for an input text field.

Table 18-24. Dependencies between tabEnabled and tabIndex for input text fields

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

tabEnabled	tabIndex	TextField included in custom tab order?	TextField included in automatic tab order?
true or undefined	undefined	no	yes
true or undefined	positive integer	yes	N/A
false	undefined	no	no
false	positive integer	no	N/A

A movie clip's `tabChildren` property affects the text fields, clips, and buttons it contains. To disable tabbing for a group of text fields, put them in a movie clip and set the clip's `tabChildren` property to `false`.

Usage

The Tab key does not work reliably in Test Movie mode unless Control Disable Keyboard Shortcuts is enabled. Be sure to test keyboard behavior in the Standalone Player, if applicable, and in the Flash Player for all targeted web browsers.

Example

The following code creates two text fields and excludes the second from the automatic tab order:

```
// Create the fields
this.createTextField("field1_txt", 1, 40, 70, 100, 20);
field1_txt.border = true;
field1_txt.type = "input";
this.createTextField("field2_txt", 2, 150, 70, 100, 20);
field2_txt.border = true;
field2_txt.type = "input";
field2_txt.tabEnabled = false;
```

See Also

Button.tabEnabled, *MovieClip*.tabChildren, *MovieClip*.tabEnabled, *TextField*.tabIndex, *TextField*.type

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Chapter 18. ActionScript Language Reference

TextField.tabIndex Property — Flash 6

specifies the text field's index in the custom tab order — ***read/write***

theField.tabIndex

Description

The integer `tabIndex` property specifies *`theField`*'s position in the custom tab order. When no text field, clip, or button has a `tabIndex` property defined, Flash determines tab order automatically and pressing the Tab key moves focus between buttons and input text fields from right to left, then top to bottom. We use the `tabIndex` property to force Tab key navigation to follow our own sequence, overriding the automatic order. For example, a custom tab order can change a two-column table of input text fields from "row-by-row" to "column-by-column" navigation. Or, it can completely curtail keyboard access to an input text field, movie clip, or button.

To create a custom tab order, we assign an integer `tabIndex` priority to each object that we want tab-accessible. At each frame rendering, ActionScript creates the tab order dynamically, according to the priorities set for any objects currently displayed. Flow moves from lowest to highest number, so when the Tab key is pressed, the object with `tabIndex` 1 is focused first, followed by `tabIndex` 2, and so on. Objects with `_visible` set to `false` are not included in the tab order.

Here we create four text fields in a two-by-two grid; then we force the tab order to flow right-to-left up the grid, one row at a time (this is the reverse of the automatic order):

```
// Create the fields
this.createTextField("row1col1_txt", 1, 40, 70, 100, 20);
row1col1_txt.border = true;
row1col1_txt.type = "input";
this.createTextField("row1col2_txt", 2, 150, 70, 100, 20);
row1col2_txt.border = true;
row1col2_txt.type = "input";
this.createTextField("row2col1_txt", 3, 40, 100, 100, 20);
row2col1_txt.border = true;
row2col1_txt.type = "input";
this.createTextField("row2col2_txt", 4, 150, 100, 100, 20);
row2col2_txt.border = true;
row2col2_txt.type = "input";
// Assign custom tab order
row2col2_txt.tabIndex = 1;
row2col1_txt.tabIndex = 2;
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

```
row1col2_txt.tabIndex = 3;
row1col1_txt.tabIndex = 4;
```

When at least one object has `tabIndex` set, any objects without an assigned `tabIndex` cannot be accessed via the Tab key. Note that `tabIndex` affects only text fields with `type` set to "`input`". By default, input text fields are tab-enabled, so the `tabEnabled` property needs to be set (to `false`) only to exclude an instance from the tab order while retaining its `tabIndex` value.

Usage

The Tab key does not work reliably in Test Movie mode unless Control Disable Keyboard Shortcuts is enabled. Be sure to test keyboard behavior in the Standalone Player, if applicable, and in the Flash Player for all targeted web browsers.

See Also

Button.tabIndex, *MovieClip*.tabIndex, *TextField*.tabEnabled, *TextField*.type

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Chapter 18. ActionScript Language Reference

TextField._target Property Flash 6

the target path of the text field, in slash syntax ***read-only***

theField._target

Description

The _target property represents the path to *theField* in a Flash 4-style slash notation string. For example, if name_txt resides on the main movie timeline, name_txt's _target property is "/name_txt". To convert the slash-style path to an object reference, use *eval( )*. To retrieve a string giving the path to a text field in dot notation, use the global *targetPath( )* function. Because slash notation is deprecated, this property is not used commonly and is supported for legacy reasons only.

See Also

Button._target, *MovieClip*._target, *targetPath( )*

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Chapter 18. ActionScript Language Reference

TextField.text Property — Flash 6

specifies the characters to be rendered in the field ***read/write***

theField.text

Description

The string `text` property specifies the characters displayed by *`theField`*. Assigning a value to the `text` property causes characters to display in *`theField`*. Retrieving the `text` property tells us the entire textual content of *`theField`*, including any user input. When text cannot fit in *`theField`*'s rectangular bounding box, the excess wraps to the next line or overflows *`theField`*, or *`theField`*'s bounding box is resized, depending on the settings of `autoSize`, `multiline`, and `wordWrap`. Any text overflow must be viewed through horizontal or vertical scrolling (see `hscroll` and `scroll`).

Text can be formatted via the *setTextFormat( )* and *setNewTextFormat( )* methods. However, reassigning the `text` property clears all previous formatting. To add text to a field without disturbing any existing formatting, use *replaceSel( )*.

See *`TextField`*`.htmlText` for a full discussion of assigning HTML-formatted text to a text field. Take heed that `htmlText` and `text` overwrite each other. Regardless of the value of the `html` property, assigning a value to `text` overwrites the value of `htmlText`, and vice versa.

Usage

In Flash 4 and Flash 5, text fields were represented by simple variables, not complete objects. Hence, it was common to assign text directly to a text field variable, as follows:

```
theField_txt = "Welcome to my site.";
```

As of Flash 6, this practice will erroneously overwrite the *TextField* instance. The correct syntax in Flash 6 is as follows:

```
theField_txt.text = "Welcome to my site.";
```

See *`TextField`*`.variable` for details.

Example

The `text` property is our main means of programmatically displaying text on screen. It's Flash's best candidate for a "hello world" example, as follows:

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

```
// Create a text field
this.createTextField("theField_txt", 1, 0, 0, 200, 20);
// Drum roll please...
theField_txt.text = "hello world!";
```

The following code echoes the value of an input text field to the Output window every 500 milliseconds. Export it in Test Movie mode and type into the text field.

```
this.createTextField("theField_txt", 1, 0, 0, 200, 20);
theField_txt.type = "input";
theField_txt.border = true;
setInterval(showInput, 500);
function showInput () {
  trace(theField_txt.text);
}
```

See Also

The *Selection* object, *TextField*.html, *TextField*.htmlText, *TextField.replaceSel( )*, *TextField*.variable

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Chapter 18. ActionScript Language Reference

TextField.textColor Property — Flash 6

sets the color for all characters in the text field — ***read/write***

theField.textColor

Description

The integer `textColor` property is a convenience property for quickly setting the color of the text (not the background color) for an entire text field with minimal fuss. It defaults to black (i.e., hex `0x000000`). The following code makes the `title_txt` field red:

```
this.createTextField("title_txt", 1, 0, 0, 200, 20);
title_txt.text = "The Story of Princess Gwendolyn";
title_txt.textColor = 0xFF0000;
```

To set the background color, use the `backgroundColor` property. To color and format text on a character-by-character basis, use *setTextFormat( )* or `htmlText` instead. Any assignment to `textColor` overwrites specific character color information contained in the field's `htmlText` property or *TextFormat* objects. Once `textColor` is set, all subsequent text appears in the specified color. Functionally, assigning `textColor` is identical to creating a *TextFormat* object with a `color` property set, then applying that format to the entire field with *setTextFormat( )* and also applying it to any new text with *setNewTextFormat( )*. The value of `textColor` is also reflected in `htmlText` if the `html` property is `true`.

See Also

The *Color* class, *TextField*.backgroundColor, *TextField*.borderColor, *TextField*.html, *TextField*.htmlText, *TextField.setNewTextFormat( )*, *TextField.setTextFormat( )*

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Chapter 18. ActionScript Language Reference

TextField.textHeight Property — Flash 6

pixel height of all the text in the text field — ***read-only***

theField.textHeight

Description

The integer `textHeight` property indicates the pixel height of the entire body of text contained by *`theField`*. It is not a measurement of the text currently displayed on screen in *`theField`*'s bounding box; rather, it is a measurement of every character in *`theField.text`*, including those characters that can be accessed only by vertical scrolling. For each line of text, the `textHeight` property adds the height specified by the font metrics of the largest character on the line, including ascenders and descenders, whether or not the character in question has an ascender or a descender or is in upper- or lowercase. Note that `textHeight` is measured in pixels, not the number of lines or the page size (see the page size calculation in the Example under *`TextField.bottomScroll`*).

For example, in 12-point Times New Roman:

- The string "an" has a `textheight` of 15.
- The string "angel" also has a `textHeight` of 15, even though it includes letters with a descender (g) and an ascender (l).
- The string "ANGEL" also has a `textHeight` of 15, even though it includes uppercase letters.
- The string "An \n angel" spans two lines, and has a `textHeight` of 30 (whether or not the lines are displayed on screen).

(Note: the font metrics listed here for Times New Roman are for one particular system; they may vary by platform.)

When the `autoSize` property is `true`, `textHeight` is always 4 pixels less than `_height`, leaving a small gutter between *`theField`*'s top and bottom borders and the displayed characters.

To determine the height of a string before assigning it to a text field, use *TextFormat.getTextExtent( )*.

Example

The following simplified scrollbar code scales a scrollbar handle in proportion to the amount of overflow in a text field. Every time the field is scrolled, the code checks the ratio of `_height` to `textHeight` to determine the scale of the handle. It positions the handle based on the `scroll` and `maxscroll` properties. To try the code, first make two rectangular movie clips named `scrollbg` and `scrollhandle`.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

```
// Create a multiline text field
this.createTextField("theField_txt", 1, 0, 0, 150, 40);
theField_txt.border = true;
theField_txt.text = "Line one \nLine two \nLine three \nLine four";
// Allow user input on multiple lines
theField_txt.type = "input";
theField_txt.multiline = true;
// Size the scrollbar background
scrollbg._width = 20;
scrollbg._height = theField_txt._height;
// Position the scrollbar background
scrollbg._x = theField_txt._x + theField_txt._width;
scrollbg._y = theField_txt._y;
// Size and position the scrollbar handle whenever the field is scrolled
theField_txt.onScroller = function () {
  // Size the scrollbar handle
  scrollhandle._width = 20;
  scrollhandle._height = theField_txt._height
                       * (theField_txt._height / theField_txt.textHeight);
  // Position the scrollbar handle
  scrollhandle._x = theField_txt._x + theField_txt._width;
  var trackSize = scrollbg._height - scrollhandle._height;
  scrollhandle._y = theField_txt._y
                  + trackSize * ((this.scroll-1)  / (this.maxscroll-1));
}
// Set the initial position of the scrollbar handle
theField_txt.onScroller();
```

See Also

TextField.autoSize, *TextField*._height, *TextField*.maxscroll, *TextField.onScroller( )*, *TextField*.scroll, *TextField*.textWidth, *TextField*._width, *TextFormat.getTextExtent( )*; the *FScrollBar* component in Appendix G

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Chapter 18. ActionScript Language Reference

TextField.textWidth Property — Flash 6

pixel width of all the text in the text field — ***read/write***

theField.textWidth

Description

The integer `textWidth` property indicates the pixel width of the longest line of text contained by *`theField`*, including those characters that be accessed only through horizontal scrolling. For each character in the longest line, the `textWidth` property adds the width specified by the font metrics for that character. Characters can vary greatly in width, depending on font size, case, and even the intrinsic width of the character itself. For example, in 12-point Times New Roman:

- The character "a" has a width of 5.
- The character "A" has a width of 9.
- The character "i" has a width of 3.
- The character "w" has a width of 9.

When the `autoSize` property is `true`, `textWidth` is always 4 pixels less than `_width`, leaving a small gutter between the field's longest line of text and the right and left borders.

To determine the width of a string before assigning it to a text field, use *TextFormat.getTextExtent( )*.

Example

The `textWidth` property can be used to create horizontal scrolling interfaces or to scale graphic elements to match the size of a line of text. The following code creates a text field and then draws a line above the field that matches its text's width. Notice that the gutter is accommodated by the line-width calculation:

```
// Create the text field
this.createTextField("theField_txt", 1, 0, 0, 200, 20);
theField_txt.text = "Welcome to my web site";
theField_txt.autoSize = true;
// Set the line thickness
this.lineStyle(3);
// Position the start point
this.moveTo(theField_txt._x + 2, theField_txt._y);
// Draw the line to the end point
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

```
this.lineTo(theField_txt._x + theField_txt.textWidth, theField_txt._y);
```

See Also

TextField.autoSize, *TextField*._height, *TextField*.scroll, *TextField*.maxhscroll, *TextField*.textHeight, *TextField*._width, *TextFormat.getTextExtent( )*

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Chapter 18. ActionScript Language Reference

TextField.type Property — Flash 6

specifies whether the field can accept user input — ***read/write***

theField.type

Description

The string `type` property determines a field's input and output behavior. It has only two possible values: "input" and "dynamic" (the default). A dynamic text field is used to display information to the user (output only); an input text field ca retrieve information from the user (input and output).

> The Flash 5 Multiline and Password settings for a text field are represented by the `multiline` and `password` properties, not by type.

When `type` is set to "input", the user can enter text into the field and we can examine the input via its `text` property (se *`TextField .text`* for a code sample). When `type` is set to "dynamic", the user cannot type into the field. Howeve characters can still be selected unless `selectable` is set to `false` explicitly. Unicode text can be entered into input t fields when Unicode is supported by the operating system.

Because data entered by the user into an input text field is always a string, we should convert it explicitly before using it i nonstring context, as demonstrated in this simple calculator example that totals two user-input text fields:

```
// Suppose myFirstInput is 5, mySecondInput is 10, and we total the fields.
// WRONG: "Adding" the fields together sets myOutput to "Total: 510"
// because the + operator is interpreted as a string concatenator.
myOutput.text = "Total: " + (myFirstInput.text + mySecondInput.text);
// RIGHT: First, convert the fields to numbers in order to get the right res
myOutput.text = "Total: " + (parseFloat(myFirstInput.text)
                           + parseFloat(mySecondInput.text));
```

Example

The following example creates a title that can neither be selected nor receive input:

```
this.createTextField("title_txt", 1, 0, 0, 200, 20);
title_txt.selectable = false;
title_txt.type = "dynamic";
title_txt.text = "Tea Brewing Around the World";
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

The following example creates an input field to allow a user to enter a name. It's customary to add a border to input text fi via the `border` property. This adds a visual cue indicating the field is editable:

```
this.createTextField("firstName_txt", 1, 0, 0, 200, 20);
firstName_txt.type = "input";
firstName_txt.border = true;
```

See Also

parseFloat( ) , *`TextField` .border* , *`TextField` .multiline* , *`TextField` .password* , *`TextFie` .restrict* , *`TextField` .selectable* , *`TextField` .text*

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Chapter 18. ActionScript Language Reference

TextField._url Property — Flash 6

the network address of the movie clip that contains the field — ***read-only***

theField._url

Description

The _url property represents the URL (Uniform Resource Locator) indicating the Internet or local disk location of *theField*._parent, as a string. The _url property is always an absolute URL, never a relative one. See *MovieClip*._url for details.

See Also

MovieClip._url, *TextField*._parent

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Chapter 18. ActionScript Language Reference

TextField.variable Property — Flash 6

associates a variable with the text property — ***read/write***

theField.variable

Description

The string `variable` property is a backward-compatibility feature that specifies a regular variable whose value will mirror the value of *theField*.`text`. This allows for direct Flash 5-style assignment of a value to a text field. The following example creates a text field object with an instance name of `firstName_txt` and associates it with the variable `userName`. Subsequent assignments to `userName` are displayed in `firstName_txt` and reflected by its `text` property.

```
this.createTextField("firstName_txt", 1, 0, 0, 200, 20);
firstName_txt.variable = "userName";
// Displays "Colin" on screen
userName = "Colin";
// Check the text property
trace(firstName_txt.text);  // Displays: Colin
```

Conversely, assignments to `firstName_txt.text` are reflected by the variable `userName`:

```
firstName_txt.text = "Bruce";
trace(userName);  // Displays: Bruce
```

The variable reference can be constructed with dot notation, allowing variables on objects and other timelines to be used:

```
firstName_txt.variable = "_level0.someClip.someVariable";
```

When exporting to Flash 5 and prior versions, the `variable` property must be set in the Flash MX authoring tool, in the Property inspector under "Var:". When coding for Flash 6, the `variable` property should be avoided in favor of the `text` property.

Note that the `variable` property stores the name of the variable, not its value. It is used only to establish the link between a regular variable and *theField*.`text`:

```
trace(firstName_txt.variable);  // Displays: userName
```

See Also

TextField.htmlText, *TextField*.text

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Chapter 18. ActionScript Language Reference

TextField._visible Property — Flash 6

whether the text field is shown or hidden — ***read/write***

theField._visible

Description

The Boolean `_visible` property indicates whether *`theField`* is currently shown (`true`) or hidden (`false`). We use `_visible` as a quick means of hiding a text field from view. Hidden fields can still be controlled via ActionScript and generally operate as normal; they simply are not displayed on screen. However, a hidden text field can receive neither focus nor user input, so its *onChanged( )*, *onKillFocus( )*, and *onSetFocus( )* events do not fire. However, the *onScroller( )* event will still fire in response to text field content changes.

The initial value of the `_visible` property is `true` for all text fields, even for text fields that are fully transparent or completely off stage. The `_visible` property changes only when it is modified explicitly by a script; think of it as a way to show and hide text fields programmatically, not as a reflection of all factors, such as position and transparency, that can affect a field's visibility.

Hiding a text field using the `_visible` property is preferred to setting it to fully transparent or moving it off stage because the Flash Player does not attempt to draw text fields with `_visible` set to `false`, thus improving rendering performance.

Example

The following code hides `theField_txt`:

```
this.createTextField("theField_txt", 1, 0, 0, 200, 20);
theField_txt.text = "You won't see this!";
theField_txt._visible = false;
```

See Also

TextField._alpha

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Chapter 18. ActionScript Language Reference

TextField._width Property — Flash 6

width of the text field's bounding box, in pixels — ***read/write***

theField._width

Description

Every text field implicitly defines a rectangular region, known as its *bounding box*, inside which text is displayed. The floating-point _width property is a nonnegative number specifying the width of a field's bounding box, in pixels. By changing _width, we resize the field's bounding box, allowing it to display more or less text. Changing a field's _width does not also affect the width of the text it contains; to change the text's point size, use a *TextFormat* object or the _yscale property. To check the width of a field's text, use textWidth. Note that in Flash 6, the autoSize property can be used to force a text field's bounding box to fit its text.

Changes made to a text field's width in the authoring tool or via _xscale are reflected by _width. To apply a stroke or a fill to a field's bounding box, use the border and background properties.

Example

```
// Set initial width to 200
this.createTextField("theField_txt", 1, 0, 0, 200, 20);
theField_txt.border = true;
theField_txt.text = "Room to grow!";
// Set the bounding box width to twice the width of the text in the fielc
theField_txt._width = theField_txt.textWidth * 2;
```

See Also

TextField.autoSize, *TextField*._height, *TextField*.textWidth, *TextField*.wordWrap, *TextField*._xscale, the *TextFormat* class

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Chapter 18. ActionScript Language Reference

TextField.wordWrap Property — Flash 6

enable automatic line-breaking for long lines — *read/write*

theField.wordWrap

Description

The Boolean wordWrap property enables (true) or disables (false) automatic line-breaking in *theField* . It a both user input and text assigned programmatically, and it defaults to false . When word-wrapping is enabled, long lin *soft wrap* —that is, any line that is wider than *theField* 's bounding box will automatically flow onto the following li is a display feature only; no carriage return or newline character appears in the source text where wrapping occurs. When wrapping is disabled, text on long lines flows beyond *theField* 's bounding box and part of the line is hidden. If wor is true , changing the field's _width causes long lines to soft wrap at a different point (i.e., the text reflows). Hard retu be added to a text field using the \n escape sequence or newline constant.

Bugs

In Flash 6, the wordWrap property must be set before any text is assigned to the text field; otherwise, wrapping may no applied. For example:

```
// Set word wrap first
theField_txt.wordWrap = true;
// Now assign text
theField_txt.text = "This is a fairly long line so it might wrap when displa
```

Example

To create a typical multi-paragraph text entry box for user input, set both wordWrap and multiline to true . For e

```
this.createTextField("theField_txt", 1, 0, 0, 200, 100);
theField_txt.type = "input";
theField_txt.border = true;
theField_txt.wordWrap = true;
theField_txt.multiline = true;
theField_txt.text = "Type input here.";
```

See Also

TextField .multiline, *TextField* ._width

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Chapter 18. ActionScript Language Reference

TextField._x Property — Flash 6

horizontal location of the text field, in pixels — *read/write*

theField._x

Description

The floating-point _x property indicates the horizontal position of *theField*'s left border. It is measured relative to one of two possible coordinate spaces:

- If *theField* resides on the main timeline, _x is measured relative to the Stage's left edge. For example, an _x of 20 indicates that *theField*'s left border is 20 pixels to the right of the Stage's left edge; -20 indicates it is 20 pixels to the left.
- If *theField* resides on a movie clip instance's timeline, _x is measured relative to the registration point of that parent instance. For example, an _x of 20 indicates that *theField*'s left border is 20 pixels to the right of its parent instance's registration point; -20 indicates 20 pixels to the left.

The _x property (along with all horizontal coordinates in Flash) increases to the right and decreases to the left. Fractional _x values are approximated in Flash with antialiasing (blurring).

If *theField* is contained by an instance that is scaled and/or rotated, the coordinate space it inhabits is also scaled and/or rotated. For example, if *theField*'s parent is scaled by 200% and rotated 90 degrees clockwise, _x increases in the downward direction, rather than to the right, and a single unit of _x is 2 pixels instead of 1.

Example

The following example uses the _x property to align one field to another (they share the same x-coordinate, so they are aligned along the same vertical line):

```
// Make the first field
this.createTextField("firstName_txt", 1, 250, 0, 200, 20);
firstName_txt.border = true;
firstName_txt.type = "input";
// Make the second field
this.createTextField("lastName_txt", 2, 0, 0, 200, 20);
lastName_txt.border = true;
lastName_txt.type = "input";
// Align the second field
lastName_txt._x = firstName_txt._x;
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

```
lastName_txt._y = firstName_txt._y + firstName_txt._height + 10;
```

The following example adds to all text fields a *slideTo( )* method that moves a text field left or right to some destination at a given speed:

```
// Method to start the slide interval
TextField.prototype.slideTo = function (xPos, pixelsPerSec) {
  var moveAmount = pixelsPerSec/20;
  if (xPos < 0) {
    moveAmount = -moveAmount;
  }
  this.slideInterval = setInterval(this, "slide", 50, moveAmount, xPos);
}
// Method to move the text field
TextField.prototype.slide = function (moveAmount, xPos) {
  this._x += moveAmount;
  if (Math.abs(this._x - xPos) < 1) {
    clearInterval(this.slideInterval);
  }
  updateAfterEvent();
}
// Make the field
this.createTextField("theField_txt", 1, 0, 0, 200, 20);
theField_txt.border = true;
theField_txt.text = "Watch me go for a ride!";
// Slide it to 500 at speed 100px/second
theField_txt.slideTo(500, 100);
```

See Also

MovieClip.globalToLocal( ), *MovieClip.localToGlobal( )*, `TextField._y`

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Chapter 18. ActionScript Language Reference

TextField._xmouse Property — Flash 6

horizontal location of mouse pointer, relative to the text field — ***read-only***

theField._xmouse

Description

The floating-point _xmouse property indicates the horizontal location of the mouse pointer's hotspot, relative to the left border of *theField*. To obtain a consistent _xmouse coordinate that is always measured relative to the Stage, use _root._xmouse instead.

See Also

MovieClip._xmouse, *TextField*._ymouse

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Chapter 18. ActionScript Language Reference

TextField._xscale Property — Flash 6

width of a text field, as a percentage of original — ***read/write***

theField._xscale

Description

The floating-point _xscale property specifies the width of *theField*'s bounding box, relative to its original width, expressed as a percentage. The original width is either the width of the field at authoring time or the width most recently specified (via either *MovieClip.createTextField( )* or _width). Text in the field is scaled only if embedFonts is true; otherwise, it is not affected by _xscale. This contrasts with _yscale, which always affects *theField*'s text scale. Note that scaling text horizontally but not vertically stretches it disproportionately. Use a *TextFormat* object for precise control over text's point size.

When the current width of *theField* is the same as its original width, _xscale is 100. An _xscale of 200 doubles *theField*'s width relative to its original width. An _xscale of 50 halves *theField*'s width relative to its original width.

When _xscale is set to a negative value and embedFonts is true, *theField* is flipped horizontally, as if across a vertical mirror running along its left edge (i.e., it becomes a mirror image of itself), after which the negative value is treated as a positive. To flip a text field horizontally without resizing it, set _xscale to -_xscale (negative _xscale).

Example

```
this.createTextField("field1_txt", 1, 0, 0, 200, 20);
field1_txt.text = "This is twice as wide.";
field1_txt.border = true;
this.createTextField("field2_txt", 2, 0, 25, 200, 20);
field2_txt.text = "This is backwards!";
field2_txt.border = true;
// Double the width of theField_txt
field1_txt._xscale *= 2;
// Create a mirror image of theField_txt
field2_txt._xscale = -field2_txt_xscale;
```

See Also

MovieClip._xscale, *TextField*._width, *TextField*._yscale, the *TextFormat* class

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Chapter 18. ActionScript Language Reference

TextField._y Property — Flash 6

vertical location of the text field, in pixels — ***read/write***

`theField._y`

Description

The floating-point `_y` property indicates the vertical position of *`theField`* 's top border. It is measured relative to one possible coordinate spaces:

- If *`theField`* resides on the main timeline, `_y` is measured relative to the Stage's top edge. For example, a `_y` of indicates that *`theField`* 's top border is 20 pixels below the Stage's top edge;-20 indicates 20 pixels above.
- If *`theField`* resides on a movie clip instance's timeline, `_y` is measured relative to the registration point of that instance. For example, a `_y` of 20 indicates that *`theField`* 's top border is 20 pixels below its parent instance's r point; -20 indicates 20 pixels above.

The `_y` property (as with all vertical coordinates in Flash) increases downwards and decreases upwards—the opposite of t Cartesian coordinate system. Fractional `_y` values are approximated in Flash with antialiasing (blurring).

If *`theField`* is contained by an instance that is scaled and/or rotated, the coordinate space it inhabits is also scaled and For example, if *`theField`* 's parent is scaled by 200% and rotated 90 degrees clockwise, `_y` increases toward the left, downward, and a single unit of `_y` is 2 pixels instead of 1.

Example

The `_y` property is often used in conjunction with `_height` to programmatically space or align text fields stacked vertic shown in the function *makeFields( )* :

```
// Try it out...
makeFields(this, 6, 5, 100, 100, 200, 20);
// Make a series of text fields
function makeFields (clip, numFields, bufferSpace, startX, startY, width, he
  // Loop to make the requested fields
  for (var i = 0; i < numFields; i++) {
    // Make the current field
    clip.createTextField("field" + i + "_txt", i+1, startX, startY, width, h
    // Store a reference to the current field
    var thisField = clip["field" + i + "_txt"];
    // Turn on border and input
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

```
      thisField.border = true;
      thisField.type = "input";
      // Position the current field if it's not the first
      if (i > 0) {
        var prevField = clip["field" + (i-1) + "_txt"];
        thisField._y = prevField._y + prevField._height + bufferSpace;
      }
    }
}
```

See Also

MovieClip.globalToLocal( ) , *MovieClip.localToGlobal( )* , *TextField* ._x

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Chapter 18. ActionScript Language Reference

TextField._ymouse Property — Flash 6

vertical location of mouse pointer, relative to the text field — ***read-only***

theField._ymouse

Description

The floating-point _ymouse property indicates the vertical location of the mouse pointer's hotspot, relative to the top border of *theField*. To obtain a consistent _ymouse coordinate that is always measured relative to the Stage, use _root._ymouse instead.

See Also

MovieClip._ymouse, *TextField*._xmouse

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Chapter 18. ActionScript Language Reference

TextField._yscale Property — Flash 6

height of a text field, as a percentage — ***read/write***

theField._yscale

Description

The floating-point _yscale property specifies the height of *theField*'s bounding box relative to its original height, expressed as a percentage. The original height is either the height of the field at authoring time or the height most recently specified via either *MovieClip.createTextField( )* or _height.

When the current height of *theField* is the same as its original height, _yscale is 100. A _yscale of 200 doubles *theField*'s height relative to its original height. A _yscale of 50 halves *theField*'s height relative to its original height. Changes to _yscale are reflected by _height.

The size of text in the field is also affected by _yscale, but it is always scaled proportionately (in relatively equal horizontal and vertical percentages) to prevent distortion. This contrasts with _xscale, which affects *theField*'s text scale only when embedFonts is true and can scale text disproportionately. Though _yscale can change the size of text, the change is not reflected by the *TextFormat* object returned by *getTextFormat( )*. For precise control over text size, use the *TextFormat* object's size property instead of *TextField*._yscale.

When _yscale is set to a negative value and embedFonts is true, *theField* is flipped vertically, as if across a horizontal mirror running through its top border (i.e., it becomes a mirror image of itself), after which the negative value is treated as a positive. To flip a text field vertically without resizing it, set _yscale to -_yscale (negative _yscale).

Example

The following code increases theField_txt's height by 500 percent:

```
this.createTextField("theField_txt", 1, 0, 0, 200, 20);
theField_txt.border = true;
theField_txt.text = "This is five times as high.";
theField_txt._yscale = 500;
// To maintain the proportions of the original bounding box,
// we can set _xscale to match _yscale:
theField_txt._xscale = 500;
```

See Also

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

MovieClip._yscale, *TextField*._height, *TextField*._xscale, the *TextFormat* class

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Chapter 18. ActionScript Language Reference

TextFormat Class Flash 6

retrieve or set a text field's visual formatting

Constructor

```
new TextFormat()
new TextFormat(font, size, color, bold, italic, underline, url,
               target, align, leftMargin, rightMargin, indent, leading)
```

Arguments

font

An optional string specifying the `font` property's initial value.

size

An optional integer specifying the `size` property's initial value.

color

An optional integer specifying the `color` property's initial value.

bold

An optional Boolean specifying the `bold` property's initial value.

italic

An optional Boolean specifying the `italic` property's initial value.

underline

An optional Boolean specifying the `underline` property's initial value.

url

An optional string specifying the `url` property's initial value.

target

An optional string specifying the `target` property's initial value.

align

An optional string specifying the `align` property's initial value.

leftMargin

An optional nonnegative integer specifying the `leftMargin` property's initial value.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

righMargin

An optional nonnegative integer specifying the `rightMargin` property's initial value.

indent

An optional nonnegative integer specifying the `indent` property's initial value.

leading

An optional nonnegative integer specifying the `leading` property's initial value.

Properties

align

Specifies horizontal paragraph alignment.

blockIndent

Specifies paragraph indentation.

bold

Boolean; specifies bold character display.

bullet

Specifies whether to add bullets to paragraphs.

color

Specifies character color.

font

Sets the font for specified characters.

indent

Specifies indentation for a paragraph's first line.

italic

Boolean; specifies italicized character display.

leading

Specifies line spacing.

leftMargin

Specifies the margin to the left of a paragraph.

rightMargin

Specifies the margin to the right of a paragraph.

size

Specifies character font size, in points.

tabStops

Specifies horizontal tab stops, in pixels.

target

Specifies the window or frame for a hypertext link.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

underline

Boolean; specifies underlined character display.

url

Specifies a hypertext link.

Methods

getTextExtent( )

Returns the pixel dimensions of a string rendered according to a given *TextFormat* object.

Description

Internally, a text field's visual formatting is described by *style profiles* stored in the Flash Player. Each profile holds format settings such as font size, font face, bolding, and margins. Every character in a text field is associated with one of these internal profiles. For example, in the text field:

Please hurry up!

two style profiles are represented: one with bold type and one without bold type. The characters *P* , *l* , *e* , *a* , *s* , and *e* are associated with the bold profile, while the remaining characters are associated with the nonbold profile. As necessary, Flash adds internal profiles to keep track of all text displayed in the Player. While ActionScript itself does not provide direct access to the Player's internal profiles, it can depict the profile for any character as a *TextFormat* object. A *TextFormat* object's properties control the display features associated with a specific character or, alternatively, report the display features common to a series of characters.

To retrieve a *TextFormat* object for an existing text field, we use the *TextField.getTextFormat( )* method. For example, here we create a text field and then retrieve a *TextFormat* object that describes the formatting of the field's first character:

```
// Create the text field
this.createTextField("theField_txt", 1, 0, 0, 150, 40);
// Assign the field some text for display
theField_txt.text = "Please hurry up!"
// Retrieve a TextFormat object for theField_txt's first character (index 0)
firstCharFormat = theField_txt.getTextFormat(0);
// Check whether the first character is bolded
trace(firstCharFormat.bold);  // Displays: "false"
```

A *TextFormat* object's usage is intimately tied to a corresponding *TextField* object. For example, the *TextFormat* object of an existing text field is obtained using *TextField.getTextFormat( )* . Throughout the following discussion, pay careful attention to which properties and methods are part of the *TextFormat* class and which are part of the *TextField* class. To set the formatting of a character or span of characters in a text field, we must create a *TextFormat* object, set its properties, and apply the format using the *TextField.setTextFormat( )* method (this is known colloquially as "applying a text format object to a text field").

The *setTextFormat( )* method optionally accepts the starting and ending character indexes over which to apply the text format object. For example:

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

```
// Create the TextFormat object
emphasisFormat = new TextFormat();
// Specify bold for the format
emphasisFormat.bold = true;
// Apply the format to the word 'Please', characters 0 to 5 (inclusive)
theField_txt.setTextFormat(0, 6, emphasisFormat);
```

Alternatively, we can set the *TextFormat* object's properties when we construct it (using `null` for properties we do not wish to set):

```
emphasisFormat = new TextFormat(null, null, null, true);
```

When text is added to a field, it is formatted according to the field's default style profile, which is represented in ActionScript by a distinct *TextFormat* object called a *new text format* . We can view a field's default style profile by using *TextField.getNewTextFormat( )* . We can modify a field's default style profile by creating a *TextFormat* object, setting its properties, and passing it to *TextField.setNewTextFormat( )* . For more details on modifying the format of new text, see *TextField.getNewTextFormat( )* and *TextField.setNewTextFormat( )* .

For a description of the paragraph and character formats available, see the property listings for *TextFormat* . Note that each property is also available through analogous HTML tags, described under *`TextField .htmlText`* and in Appendix E. The default settings for an existing text field's *TextFormat* object are listed in Table 18-25.

Table 18-25. Default TextFormat property values

Property	Default value	Datatype
`bullet`	`false`	boolean
`tabStops`	empty object	object
`blockIndent`	0	number
`leading`	0	number
`indent`	0	number
`rightMargin`	0	number
`leftMargin`	0	number
`align`	`"left"`	string
`underline`	`false`	boolean
`italic`	`false`	boolean
`bold`	`false`	boolean
`target`	`""` (empty string)	string
`url`	`""` (empty string)	string
`color`	0	number
`size`	12	number
`font`	`"Times New Roman"`	string

Usage

Writing to the `text` property of a *TextField* object destroys any custom formatting associated with the field. Use the *TextField.replaceSel( )* method to add text to a field while retaining its formatting.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

See Also

TextField.getNewTextFormat( ) , *TextField.getTextFormat( )* , `TextField .htmlText` , *TextField.replaceSel ( )* , *TextField.setNewTextFormat( )* , *TextField.setTextFormat( )* , the *TextField* class

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Chapter 18. ActionScript Language Reference

TextFormat.align Property — Flash 6

specifies horizontal paragraph alignment — ***read/write***

theFormat.align

Description

The string `align` property specifies whether text is positioned at the "left", "right", or "center" of a text field. It applies to paragraphs, which are spans of text separated by line break characters specified in code (`\n`, `<BR>`, `<P>`, or `newline`). To set the alignment for an entire text field, apply the text format by invoking *TextField.setTextFormat( )* without specifying any index arguments:

```
this.createTextField("theField_txt", 1, 0, 0, 200, 60);
theField_txt.border = true;
theField_txt.text = "This is paragraph one.\nThis is paragraph two."
// Create a TextFormat object
format = new TextFormat();
format.align = "center";
// Center the whole field
theField_txt.setTextFormat(format);
```

To set the alignment for the first paragraph in a text field, apply the format to the character at index 0:

```
theField_txt.setTextFormat(0, format);
```

To set the alignment for any subsequent paragraphs, apply the format to the character immediately following the newline character:

```
// Center the second paragraph
theField_txt.setTextFormat(23, format);
```

To set the alignment for a range of paragraphs, apply the format using start and end index arguments that include the desired paragraphs:

```
theField_txt.setTextFormat(0, 24, format);
```

If any paragraph wraps to the next line, the specified format range must include the subsequent line's first character; otherwise, the format will not apply to the wrapped line.

If automatic word-wrapping is disabled for a text field and the text of a paragraph exceeds the width of the field, the text will be left-aligned, regardless of the setting of `align`.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

The default `align` value (as contained in the *TextFormat* object returned by *TextField.getTextFormat( )*) for a text field that contains text is "left". For a text field without any text, the `align` value is `null`. For a completely new *TextFormat* object, it is also `null`, indicating that `align` will not be set when the format is applied to a text field:

```
trace(anyUnformattedField_txt.getTextFormat().align);  // Displays: "left"
var format = new TextFormat();
trace(format.align);                                   // Displays: "null"
```

See Also

TextField.wordWrap, *TextFormat*.blockIndent, *TextFormat*.indent, *TextFormat*.leftMargin, *TextFormat*.rightMargin

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Chapter 18. ActionScript Language Reference

TextFormat.blockIndent Property — Flash 6

specifies paragraph indentation — ***read/write***

theFormat.blockIndent

Description

The integer `blockIndent` property specifies the distance a paragraph is indented from the text field's left border, in pi
Paragraphs are spans of text separated by line characters specified in code (`\n` , `<BR>` , or `newline`). To set the inden
for an entire text field, invoke *TextField.setTextFormat( )* without specifying any index arguments:

```
this.createTextField("theField_txt", 1, 0, 0, 200, 60);
theField_txt.border = true;
theField_txt.text = "This is paragraph one.\nThis is paragraph two."
// Create a TextFormat object
format = new TextFormat();
format.blockIndent = 30;
// Block-indent the whole field
theField_txt.setTextFormat(format);
```

To set the indentation for the first paragraph in a text field, apply the format to the character at index 0:

```
theField_txt.setTextFormat(0, format);
```

To set the indentation for any subsequent paragraphs, apply the format to the character immediately following the newline
character:

```
// Indent the second paragraph
theField_txt.setTextFormat(23, format);
```

To set the indentation for a range of paragraphs, apply the format using start and end index arguments that include the des
paragraphs:

```
theField_txt.setTextFormat(0, 24, format);
```

If any paragraph wraps to the next line, the specified format range must include the subsequent line's first character; other
the format will not apply to the wrapped line.

If automatic word-wrapping is disabled for a text field, large values for `blockIndent` can force text to overflow the te
field's right border. If word-wrapping is enabled, large values for `blockIndent` cause lines to wrap. The `blockInd`
and `leftMargin` properties are cumulative; for example, if `leftMargin` is 15 and `blockIndent` for a paragrap
10, that paragraph will be indented a total of 25 pixels. Hence, the `leftMargin` property can be thought of as a field l

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

tool, while `blockIndent` can be thought of as a paragraph layout tool.

The default `blockIndent` value (as contained in the *TextFormat* object returned by *TextField.getTextFormat( )*) for a field that contains text is 0. For a text field without any text, the `blockIndent` value is `null` . For a completely new *TextFormat* object, it is also `null` , indicating that `blockIndent` will not be set when the format is applied to a text f

```
trace(anyUnformattedField_txt.getTextFormat().blockIndent);  // Displays: 0
var format = new TextFormat();
trace(format.blockIndent);                                   // Displays: "n
```

See Also

TextField .wordWrap , *TextFormat* .align , *TextFormat* .indent , *TextFormat* .leftMa
, *TextFormat* .rightMargin

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Chapter 18. ActionScript Language Reference

TextFormat.bold Property — Flash 6

Boolean; specifies bold character display — ***read/write***

theFormat.bold

Description

The Boolean `bold` property specifies whether characters are displayed in a **bold** typeface (`true`) or a regular (nonbold) typeface (`false`).

To set bolding for an entire text field, invoke *TextField.setTextFormat( )* without specifying any index arguments:

```
this.createTextField("theField_txt", 1, 0, 0, 200, 20);
theField_txt.border = true;
theField_txt.text = "Press any key."
// Create a TextFormat object
format = new TextFormat();
format.bold = true;
// Bold the whole field
theField_txt.setTextFormat(format);
```

To set bolding for a single character, apply the format with a single index argument:

```
theField_txt.setTextFormat(1, format); // Bold the second character ("r")
```

To set bolding for a range of characters, apply the format using start and end index arguments:

```
theField_txt.setTextFormat(6, 9, format);  // Bold the characters at
                                           // indexes 6, 7, and 8 ("any")
```

The default `bold` value (as contained in the *TextFormat* object returned by *TextField.getTextFormat( )*) for a text field that contains text is `false`. For a text field without any text, the `bold` value is `null`. For a completely new *TextFormat* object, it is also `null`, indicating that `bold` will not be set when the format is applied to a text field:

```
trace(anyUnformattedField_txt.getTextFormat().bold);  // Displays: "false"
var format = new TextFormat();
trace(format.bold);                                   // Displays: "null"
```

See Also

TextFormat.italic

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Chapter 18. ActionScript Language Reference

TextFormat.bullet Property Flash 6

specifies whether to add bullets to paragraphs ***read/write***

theFormat.bullet

Description

The Boolean `bullet` property can create a bulleted list (equivalent to HTML's `<UL>` tag) from the paragraphs in a text field. When `bullet` is `true` , the first line of each formatted paragraph (any span of text ending with a line break character) starts with a small circle icon and is indented. If automatic word-wrapping is enabled for the text field, subsequent lines in the paragraph are also indented, provided that they fall within the range of the format. The circular bullet symbol cannot be customized. Nested bulleted lists are not supported.

To convert an entire text field into a bulleted list, invoke *TextField.setTextFormat( )* without specifying any index arguments:

```
this.createTextField("theField_txt", 1, 0, 0, 200, 60);
theField_txt.border = true;
theField_txt.text = "Item one.\nItem two.\nItem three."
// Create a TextFormat object
format = new TextFormat();
format.bullet = true;
// Make the whole field bulleted
theField_txt.setTextFormat(format);
```

To add a bullet to the first paragraph in a text field, apply the format to the character at index 0:

```
theField_txt.setTextFormat(0, format);
```

To add a bullet to any subsequent paragraphs, apply the format to the character immediately following the newline character:

```
// Make the second paragraph bulleted
theField_txt.setTextFormat(10, format);
```

To add bullets to a range of paragraphs, apply the format using start and end index arguments that include the desired paragraphs:

```
// Use bullets with only the second and third paragraphs
theField_txt.setTextFormat(10, 21, format);
```

If any paragraph wraps to the next line, the specified format range must include the subsequent line's first character;

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

otherwise, the format will not apply to the wrapped line.

The default `bullet` value (as contained in the *TextFormat* object returned by *TextField.getTextFormat( )*) for a text field that contains text is `false` . For a text field without any text, the `bullet` value is `null` . For a completely new *TextFormat* object, it is also `null` , indicating that `bullet` will not be set when the format is applied to a text field:

```
trace(anyUnformattedField_txt.getTextFormat().bullet);  // Displays: "false"
var format = new TextFormat();
trace(format.bullet);                                   // Displays: "null"
```

See Also

TextField .wordWrap , *TextFormat* .blockIndent

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

TextFormat.color Property

Flash 6

specifies character color ***read/write***

theFormat.color

Description

The integer `color` property specifies the color of characters using RGB hexadecimal triplet notation—`0x` *RRGGBB* , where *RR* , *GG* , and *BB* are two-digit hex numbers representing Red, Green, and Blue. For example, the following sets `color` to red:

```
theFormat.color = 0xFF0000;
```

For a complete discussion of RGB triplet notation, see the *Color* class.

To set the color of an entire text field, invoke *TextField.setTextFormat( )* without specifying any index arguments:

```
this.createTextField("theField_txt", 1, 0, 0, 200, 20);
theField_txt.border = true;
theField_txt.text = "Play with color."
// Create a TextFormat object
format = new TextFormat();
format.color = 0xFF0000;
// Make the whole field red
theField_txt.setTextFormat(format);
```

Alternatively, use the convenient *TextField* `.textColor` property directly on the field, without a *TextFormat* object:

```
theField_txt.textColor = 0xFF0000;
```

To set the color of a single character, apply the format with a single index argument:

```
theField_txt.setTextFormat(0, format); // Make the first character red
```

To set the color of a range of characters, apply the format with start and end index arguments:

```
theField_txt.setTextFormat(10, 15, format);  // Set red for characters 10,
                                             // 11, 12, 13, and 14 ("color")
```

The default `color` value (as contained in the *TextFormat* object returned by *TextField.getTextFormat( )*) for a text field that contains text is 0 (black). For a text field without any text, the `color` value is `null` . For a completely new

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

TextFormat object, it is also `null` , indicating that `color` will not be set when the format is applied to a text field:

```
trace(anyUnformattedField_txt.getTextFormat().color);  // Displays: 0
var format = new TextFormat();
trace(format.color);                                   // Displays: "null"
```

See Also

The *Color* class , *`TextField`* `.textColor`

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Chapter 18. ActionScript Language Reference

TextFormat.font Property — Flash 6

sets the font for specified characters — ***read/write***

theFormat.font

Description

The string `font` property specifies the name of the font in which characters should be displayed. For text fields with font embedding disabled, `font` should exactly match the name of a font on the playback system. For example, on Windows, t following code sets `theField_txt` 's font to Arial:

```
this.createTextField("theField_txt", 1, 0, 0, 200, 20);
theField_txt.embedFonts = false;
theField_txt.text = "This text is in Arial."
format = new TextFormat();
format.font = "Arial";
theField_txt.setTextFormat(format);
```

For a list fonts installed on the playback system, use *TextField.getFontList( )* . When embedded fonts are enabled for a tex `font` property must match one of the following:

- The linkage identifier of a font symbol exported with the movie
- The name of a font exported with an author-time text field

For more information on embedding fonts with a movie see *`TextField`* `.embedFonts` .

To set the font for an entire text field, invoke *TextField.setTextFormat( )* without specifying any index arguments:

```
// Make the whole field Arial
theField_txt.setTextFormat(format);
```

To set the font for a single character, apply the format with a single index argument:

```
theField_txt.setTextFormat(0, format); // Make the first character Arial
```

To set the font for a range of characters, apply the format with start and end index arguments:

```
theField_txt.setTextFormat(0, 4, format);  // Make the word "This" Arial
```

The default `font` value (as contained in the *TextFormat* object returned by *TextField.getTextFormat( )*) for a text field t contains text is "Times New Roman". For a text field without any text, the `font` value is `null` . For a completely new *T*

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

object, it is also `null` , indicating that `font` will not be set when the format is applied to a text field:

```
trace(anyUnformattedField_txt.getTextFormat().font); // Displays: "Times New
var format = new TextFormat();
trace(format.font);                                  // Displays: "null"
```

For text fields with font-embedding disabled, the `font` property can contain a comma-separated list of fonts. Flash will r text field with the first available font in the list. For instance, "Arial,Helvetica,_sans" (no spaces between font names!) cau Player to search for the font Arial, then Helvetica (if Arial is not found), then _sans (if neither Arial nor Helvetica is found an undocumented feature of the Flash Player; however, it mirrors the behavior of HTML text font-rendering in browsers.

See Also

`TextField` `.embedFonts` , *TextField.getFontList( )*

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Chapter 18. ActionScript Language Reference

TextFormat.getTextExtent( ) Method Flash 6

returns the pixel dimensions of a string rendered according to a given TextFormat object

theFormat.getTextExtent(*text*)

Arguments

text

The string to be measured in this character format.

Returns

An object with `height` and `width` properties.

Description

The *getTextExtent( )* method returns an object whose `height` and `width` properties tell us the anticipated size of `text` formatted according to `theFormat` . It is used not to measure text already assigned to a text field but to determine the dimensions of any arbitrary string in a particular character format. We make these theoretical measurements when generat dynamic interfaces. For example, a string that is found to be too large in a certain format can be reformatted to fit availabl space. The following Example truncates text with an ellipsis (...) if the text does not fit in a text field.

The *getTextExtent( )* method measures only character formats; it ignores the effect of all paragraph formats (word-wrappi line breaks, margins, bullets, indent, block indent, etc). For example, the following code with indentation:

```
f = new TextFormat();
f.indent = 30;  // Add indentation.
trace(f.getTextExtent("test").width);
```

yields the same result as this code, with no indentation:

```
f = new TextFormat();
trace(f.getTextExtent("test").width);
```

Example

The following code adds a new method, called *setEllipsisText( )* , to all text fields. Using *getTextExtent( )* , the method checks the width of a string before assigning it to a text field. If the string is too wide to fit in the field, the method crops t string and adds an ellipsis (...):

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

```
/*
 * Method: TextField.setEllipsisText()
 *   Desc: Truncates text to fit in a text field, and adds
 *         an ellipsis if necessary
 * Params: theText     Text to assign to the text field
 */
TextField.prototype.setEllipsisText = function (theText) {
  // Retrieve the format for newly added text
  var format = this.getNewTextFormat();
  // Initialize the substring being assigned to the field
  var theSubstring = theText;
  // Check how wide the text display area is. Subtract 8 to account
  // for the gutter between the text and the text field's border.
  var textRegion = this._width - 8;
  // Assume the string is too wide to fit
  var tooWide = true;
  // Check if our assumption is correct
  for (var i = theSubstring.length - 1; tooWide =  = true && i > 0; i--) {
    // If the string is too wide to fit in the field...
    if (format.getTextExtent(theSubstring).width > textRegion) {
       // ...remove a character, add an ellipsis, then check again.
       theSubstring = theText.substr(0, i) + "...";
    } else {
       // ...otherwise, the string will fit, so stop checking.
       tooWide = false;
    }
  }
  // Assign the corrected string to the text field
  this.text = theSubstring;
}
// Sample use of setEllipsisText()
this.createTextField("theField_txt", 1, 100, 100, 150, 60);
theField_txt.border = true;
theField_txt.setEllipsisText("If this text is too long it will be truncated.
```

See Also

TextField .border, *TextField* ._height, *TextField* .textHeight, *TextField* .textWidth, *TextField* ._width, *TextFormat* .size

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Chapter 18. ActionScript Language Reference

TextFormat.indent Property Flash 6

specifies indentation for a paragraph's first line ***read/write***

theFormat.indent

Description

The integer `indent` property specifies the distance a paragraph's first line is indented from the text field's left border, in pixels. Paragraphs are spans of text separated by line break characters specified in code (`\n` , `<BR>` , or `newline`). To set the indentation for all paragraphs in a text field, invoke *TextField.setTextFormat( )* without specifying any index arguments:

```
this.createTextField("theField_txt", 1, 0, 0, 200, 60);
theField_txt.border = true;
theField_txt.text = "This is paragraph one.\nThis is paragraph two."
// Create a TextFormat object
format = new TextFormat();
format.indent = 15;
// Indent the whole field
theField_txt.setTextFormat(format);
```

To set the indentation for the first paragraph in a text field, apply the format to the character at index 0:

```
theField_txt.setTextFormat(0, format);
```

To set the indentation for any subsequent paragraphs, apply the format to the character immediately following the newline character:

```
// Indent the second paragraph:
theField_txt.setTextFormat(23, format);
```

To set the indentation for a range of paragraphs, apply the format using start and end index arguments that include the desired paragraphs:

```
theField_txt.setTextFormat(0, 24, format);
```

If automatic word-wrapping is disabled for a text field, large values for `indent` can force text to overflow the text field's right border. If word-wrapping is enabled, large values for `indent` cause lines to wrap.

The default `indent` value (as contained in the *TextFormat* object returned by *TextField.getTextFormat( )*) for a text field that contains text is 0. For a text field without any text, the `indent` value is `null` . For a completely new *TextFormat* object, it is also `null` , indicating that `indent` will not be set when the format is applied to a text field:

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

```
trace(anyUnformattedField_txt.getTextFormat().indent);  // Displays: 0
var format = new TextFormat();
trace(format.indent);                                   // Displays: "null"
```

See Also

TextField .wordWrap, *TextFormat* .align, *TextFormat* .blockIndent

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Chapter 18. ActionScript Language Reference

TextFormat.italic Property Flash 6

Boolean; specifies italicized character display ***read/write***

theFormat.italic

Description

The Boolean `italic` property specifies whether characters are displayed in an *italic* typeface (`true`) or a regular (nonitalic) typeface (`false`).

To set italics for an entire text field, invoke *TextField.setTextFormat( )* without specifying any index arguments:

```
this.createTextField("theField_txt", 1, 0, 0, 200, 20);
theField_txt.border = true;
theField_txt.text = "What a nice hat!"
// Create a TextFormat object
format = new TextFormat();
format.italic = true;
// Italicize the whole field
theField_txt.setTextFormat(format);
```

To set italics for a single character, apply the format with a single index argument:

```
theField_txt.setTextFormat(5, format); // Italicize the sixth character ("a"
```

To set italics for a range of characters, apply the format with start and end index arguments:

```
theField_txt.setTextFormat(7, 11, format);  // Italicize the characters at
                                            // indexes 7, 8, 9 and 10 ("nice
```

The default `italic` value (as contained in the *TextFormat* object returned by *TextField.getTextFormat( )*) for a text fiel that contains text is `false` . For a text field without any text, the `italic` value is `null` . For a completely new *TextFormat* object, it is also `null` , indicating that `italic` will not be set when the format is applied to a text field:

```
trace(anyUnformattedField_txt.getTextFormat().italic);  // Displays: "false"
var format = new TextFormat();
trace(format.italic);                                   // Displays: "null"
```

See Also

TextFormat .bold

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Chapter 18. ActionScript Language Reference

TextFormat.leading Property — Flash 6

specifies line spacing — ***read/write***

theFormat.leading

Description

The integer leading property specifies the amount of vertical space between lines of text. It applies to paragraphs, which are spans of text separated by line break characters specified in code (\n ,
 , or newline). When leading is 0, the font's metrics determine the line spacing. When leading is any positive integer, it specifies the extra space to add between lines, in pixels. Negative values for leading cannot be set via ActionScript but can be set during authoring with the Properties inspector for text.

To set the leading for an entire text field, invoke *TextField.setTextFormat( )* without specifying any index arguments:

```
this.createTextField("theField_txt", 1, 0, 0, 50, 60);
theField_txt.border = true;
theField_txt.text = "This is paragraph one.\nThis is paragraph two."
theField_txt.wordWrap = true;
// Create a TextFormat object
format = new TextFormat();
format.leading = 5;
// Apply leading to the whole field
theField_txt.setTextFormat(format);
```

To set the leading for the first paragraph in a text field, apply the format to the character at index 0:

```
theField_txt.setTextFormat(0, format);
```

To set the leading for any subsequent paragraphs, apply the format to the character immediately following the newline character:

```
theField_txt.setTextFormat(23, format);
```

To set the leading for a range of paragraphs, apply the format using start and end index arguments that include the desired paragraphs:

```
theField_txt.setTextFormat(0, 24, format);
```

If any paragraph wraps to the next line, the specified format range must include the subsequent line's first character; otherwise, the format will not apply to the wrapped line.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

The default `leading` value (as contained in the *TextFormat* object returned by *TextField.getTextFormat( )*) for a text field that contains text is 0. For a text field without any text, the `leading` value is `null` . For a completely new *TextFormat* object, it is also `null` , indicating that `leading` will not be set when the format is applied to a text field:

```
trace(anyUnformattedField_txt.getTextFormat().leading);  // Displays: 0
var format = new TextFormat();
trace(format.leading);                                   // Displays: "null"
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Chapter 18. ActionScript Language Reference

TextFormat.leftMargin Property Flash 6

specifies the margin to the left of a paragraph ***read/write***

theFormat.leftMargin

Description

The integer `leftMargin` property specifies the horizontal distance between the left border of a text field and the left ec of a paragraph, in pixels. Negative values for `leftMargin` are converted to 0 by ActionScript. Note that regardless of t value of `leftMargin` , Flash automatically adds a small gutter between text and a text field's border. There is no way t align text perfectly flush with the border of a text field. The `leftMargin` and `rightMargin` properties can be used simultaneously on the same paragraph.

To set the left margin for an entire text field, invoke *TextField.setTextFormat( )* without specifying any index arguments:

```
this.createTextField("theField_txt", 1, 0, 0, 200, 40);
theField_txt.border = true;
theField_txt.text = "This is paragraph one.\nThis is paragraph two."
theField_txt.wordWrap = true;
// Create a TextFormat object
format = new TextFormat();
format.leftMargin = 10;
// Apply a 10-pixel left margin to the whole field
theField_txt.setTextFormat(format);
```

To set the left margin for the first paragraph in a text field, apply the format to the character at index 0:

```
theField_txt.setTextFormat(0, format);
```

To set the left margin for any subsequent paragraphs, apply the format to the character immediately following the newline character:

```
theField_txt.setTextFormat(23, format);
```

To set the left margin for a range of paragraphs, apply the format using start and end index arguments that include the desi paragraphs:

```
theField_txt.setTextFormat(0, 24, format);
```

If any paragraph wraps to the next line, the specified format range must include the subsequent line's first character; otherwise, the format will not apply to the wrapped line.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

The `blockIndent` and `leftMargin` properties are cumulative; for example if `leftMargin` is 15 and `blockIndent` for a paragraph is 10, that paragraph will be indented a total of 25 pixels. Hence, the `leftMargin` property can be thought of as a field layout tool, while `blockIndent` can be thought of as a paragraph layout tool.

The default `leftMargin` value (as contained in the *TextFormat* object returned by *TextField.getTextFormat( )*) for a t field that contains text is 0. For a text field without any text, the `leftMargin` value is `null` . For a completely new *TextFormat* object, it is also `null` , indicating that `leftMargin` will not be set when the format is applied to a text fie

```
trace(anyUnformattedField_txt.getTextFormat().leftMargin);  // Displays: 0
var format = new TextFormat();
trace(format.leftMargin);                                   // Displays: "nu
```

See Also

TextFormat .blockIndent , *TextFormat* .rightMargin

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Chapter 18. ActionScript Language Reference

TextFormat.rightMargin Property — Flash 6

specifies the margin to the right of a paragraph — ***read/write***

theFormat.rightMargin

Description

The integer rightMargin property specifies the minimum horizontal space between the right border of a text field and right edge of a paragraph, in pixels. Negative values for rightMargin are converted to 0 by ActionScript. Note that regardless of the value of rightMargin , Flash automatically adds a small gutter between text and a text field's border. There is no way to align text perfectly flush with the border of a text field. The rightMargin and leftMargin properties can be used simultaneously on the same paragraph.

For example usage, see *TextFormat* .leftMargin .

The default rightMargin value (as contained in the *TextFormat* object returned by *TextField.getTextFormat( )*) for a field that contains text is 0. For a text field without any text, the rightMargin value is null . For a completely new *TextFormat* object, it is also null , indicating that rightMargin will not be set when the format is applied to a text f

```
trace(anyUnformattedField_txt.getTextFormat().rightMargin);  // Displays: 0
var format = new TextFormat();
trace(format.rightMargin);                                   // Displays: "n
```

See Also

TextFormat .blockIndent , *TextFormat* .leftMargin

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Chapter 18. ActionScript Language Reference

TextFormat.size Property — Flash 6

specifies character font size, in points — ***read/write***

theFormat.size

Description

The integer size property specifies character font size, in points (1/72 inch). By increasing the size property, we incr the size of on-screen text. Text in a font size that exceeds the dimensions of a text field's bounding box will appear cut off size a text field's bounding box, use the *TextField*._height, *TextField*._width, or *TextField*.autoSize properties.

To set the font size for an entire text field, invoke *TextField.setTextFormat( )* without specifying any index arguments:

```
this.createTextField("theField_txt", 1, 0, 0, 250, 40);
theField_txt.border = true;
theField_txt.text = "This text is quite large.";
// Create a TextFormat object
format = new TextFormat();
format.size = 26;
// Set the whole field to 26 pt
theField_txt.setTextFormat(format);
```

To set the font size for a single character, apply the format with a single index argument:

```
theField_txt.setTextFormat(5, format); // Set the sixth character ("t") to 2
```

To set the font size for a range of characters, apply the format with start and end index arguments:

```
theField_txt.setTextFormat(19, 24, format);  // Set size for
                                             // characters 19-23 ("large")
```

The default size value (as contained in the *TextFormat* object returned by *TextField.getTextFormat( )*) for a text field t contains text is 12. For a text field without any text, the size value is null. For a completely new *TextFormat* object, also null, indicating that size will not be set when the format is applied to a text field:

```
trace(anyUnformattedField_txt.getTextFormat().size);  // Displays: 12
var format = new TextFormat();
trace(format.size);                                   // Displays: "null"
```

Note that when font embedding is enabled for a text field, that field's text is always antialiased. Therefore, using embedde fonts with sizes smaller than 10-point is not recommended because antialiased text becomes unreadable below 10-point in

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

fonts. To prevent a font from antialiasing, use device fonts (i.e., system fonts) by setting *TextField* .embedFonts false . Device fonts are never antialiased. Pixel fonts that display clearly at small font sizes are available commercially sites such as http://www.miniml.com and http://www.fontsforflash.com.

Usage

Technically, a font's size is measured in pixels, not points. However, because most desktop displays render at 72 pixels inch, we normally can use the terms "pixel" and "point" interchangeably. But on devices with non-72 ppi display, Flash d not automatically perform any coordinate space conversions—a size of 10 always means the font is 10 pixels high, even i screen renders, say, 200 pixels per inch.

See Also

TextField .embedFonts , *TextField* ._yscale , *TextFormat* .font , *TextFormat.getTextExtent(*

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Chapter 18. ActionScript Language Reference

TextFormat.tabStops Property — Flash 6

specifies horizontal tab stops, in pixels — ***read/write***

theFormat.tabStops

Description

The array `tabStops` property specifies a list of *tab stops* for a paragraph, in pixels. As in word processing, a tab stop is position at which items separated by the Tab character are aligned (i.e., the column where the insertion point jumps when Tab key is pressed). In ActionScript, tab stops are used for tabular data entry and to produce simple tables, as shown in the following Example.

Each element in the `tabStops` array is measured from the left border of the text field, not from the previous tab stop. F example, a `tabStops` value of `[50, 120]` indicates two tab stops, one at 50 pixels and one at 120 pixels. To add a t character to a string, type the Tab key directly, use the escape sequence "`\t`", or use *String.fromCharCode(9)* . For convenience, we can assign a tab character to a variable:

```
var tab = String.fromCharCode(9);
```

A tab character always causes the insertion point to move ahead to the next available tab stop, even if previous tab stops a skipped. For example, if `tabStops` is set to `[5, 100]` for a text field that contains:

```
"Colin" + String.fromCharCode(9) + "Moock"
```

then "Moock" will appear 100 pixels right of the text field's left border. The word "Colin" is wider than 5 pixels, so the fir tab stop is ignored and the next available one is used. Remember, tab stops are set in pixels, not characters. Average chara in a 10-point font are about 8 pixels wide.

To set the tab stops for an entire text field, invoke *TextField.setTextFormat( )* without specifying any index arguments:

```
this.createTextField("theField_txt", 1, 0, 0, 250, 40);
theField_txt.border = true;
var tab = String.fromCharCode(9);
theField_txt.text = "One" + tab + "Two" + tab + "Three\n"
                  + "Four" + tab + "Five" + tab + "Six";
// Create a TextFormat object
format = new TextFormat();
format.tabStops = [50, 130];
// Apply the tab stops
theField_txt.setTextFormat(format);
```

To set the tab stops for the first paragraph in a text field, apply the format to the character at index 0:

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

```
theField_txt.setTextFormat(0, format);
```

To set the tab stops for any subsequent paragraphs, apply the format to the character immediately following the newline character:

```
theField_txt.setTextFormat(14, format);
```

To set the tab stops for a range of paragraphs, apply the format using start and end index arguments that include the desire paragraphs:

```
theField_txt.setTextFormat(0, 15, format);
```

If any paragraph wraps to the next line, the specified format range must include the subsequent line's first character; other the format will not apply to the wrapped line.

By default, a text field has tab stops every `(4 * average character width )` pixels and `tabStops` is an en array. For a text field without any text, `tabStops` is `null` . For a completely new *TextFormat* object, it is also `null` , indicating that `tabStops` will not be set when the format is applied to a text field:

```
trace(anyUnformattedField_txt.getTextFormat().tabStops);  // Displays: nothi
                                                          // (an empty array
var format = new TextFormat();
trace(format.tabStops);                                   // Displays: "null
```

Example

The following code adds to all *TextField* objects a method that programmatically places an array of text in a table structur using a the `tabStops` property. Take note that it relies on the custom *appendText( )* method listed under *TextField.replaceSel( )* . Both custom methods are available from the Code Depot.

```
/*
 * Method: TextField.makeTable
 *   Desc: Formats an array of text as a table.
 * Params: tabStops     Array containing the positions of the columns,
 *                      starting with column 2, not column 1. Text in each t
 *                      cell cannot wrap, so text must not exceed each tab s
 *         data         The array to format.
 */
TextField.prototype.makeTable = function (tabstops, data) {
  // Store the tab character in a variable
  var tab = "\t";
  // Calculate number of columns
  var numCols = tabstops.length;
  // Calculate number of rows
  var numRows = Math.ceil(data.length/numCols);
  // Initialize our text variable that will eventually be assigned to the fi
  var newText = "";
  // Add tab characters between each column,
  // and add newline characters after each row.
  // For each row...
  for (var i = 0; i < numRows; i++) {
    // ...cycle through each column.
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

```
    for (var j = 0; j < numCols; j++) {
      // Add each item in the array to our text
      newText += data[i*numCols + j];
      // If we're done with all the elements in the array, quit the loop
      if (i*numCols + j =  = data.length - 1) {
        break;
      }
      // If we're not on the last column...
      if (j < numCols - 1) {
        // ...add a tab character.
        newText += tab;
      } else {
        // ...otherwise add a newline character.
        newText += "\n"
      }
    }
  }
  // Finally, we have to apply the tab stops using a TextFormat object.
  // Create the TextFormat object, but keep the old tabStops property
  // around so we can put it back when we're done. Hence, future text added
  // will use the field's original tab stops, not our table's tab stops.
  format = this.getNewTextFormat();
  var oldTabStops = format.tabStops;
  format.tabStops = tabstops;
  // Apply the tab stops.
  this.setNewTextFormat(format);
  // Insert our table.
  this.appendText("\n" + newText + "\n");
  // Restore the original tab stops.
  format.tabStops = oldTabStops;
  this.setNewTextFormat(format);
}
// Example use. Create an array that will become a table.
var tableData = ["one", "two", "three", "four", "five", "six", "seven"];
// Create a text field to hold the table.
this.createTextField("theField_txt", 1, 0, 0, 250, 60);
theField_txt.border = true;
// Add some text and the table.
theField_txt.appendText("Here is the table:");
theField_txt.makeTable([50, 125, 200], tableData);
theField_txt.appendText("Isn't it pretty?");
```

See Also

String.fromCharCode ( ) , *TextField.replaceSel ( )* , *TextFormat* .blockIndent , *TextFormat* .leftMarg

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Chapter 18. ActionScript Language Reference

TextFormat.target Property Flash 6

specifies the window or frame for a hypertext link ***read/write***

theFormat.target

Description

The string target property specifies the name of the browser window or frame into which to load the document at *theFormat* .url . It can be a custom name or one of the four presets: "_blank ", "_parent ", "_self ", or "_top ". If *theFormat* .url is not specified, target has no effect. Take special note that url , and hence also target , works only when *TextField.* html is set to true for the text field.

The target property must be used in conjunction with the url property. To set the target property for an entire text field, invoke *TextField.setTextFormat( )* without specifying any index arguments:

```
this.createTextField("theField_txt", 1, 0, 0, 250, 40);
theField_txt.border = true;
theField_txt.text = "Visit www.moock.org.";
theField_txt.html = true;   // REQUIRED!!
// Create a TextFormat object
format = new TextFormat();
format.url = "http://www.moock.org/";
format.target = "_blank";  // Open link in new window
format.underline = true;   // Add underline manually
// Apply the hypertext link
theField_txt.setTextFormat(format);
```

To add a link to a single character, apply the format with a single index argument:

```
theField_txt.setTextFormat(0, format); // Link the first character, "V"
```

To add a link to a range of characters, apply the format with start and end index arguments:

```
// Link the text: "www.moock.org"
theField_txt.setTextFormat(6, theField_txt.length, format);
```

The default target value (as contained in the *TextFormat* object returned by *TextField.getTextFormat( )*) for a text field that contains text is the empty string (""). For a text field without any text, the target value is null . For a completely new *TextFormat* object, it is also null , indicating that target will not be set when the format is applied to a text field:

```
trace(anyUnformattedField_txt.getTextFormat().target);  // Displays nothing
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

```
var format = new TextFormat();
trace(format.target);                                    // Displays: "null"
```

See Also

TextField. html,*TextFormat* .url

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Chapter 18. ActionScript Language Reference

TextFormat.underline Property Flash 6

Boolean; specifies underlined character display ***read/write***

theFormat.underline

Description

The Boolean `underline` property specifies whether characters are displayed with a line beneath them (`true`) or with underline (`false`).

To underline an entire text field, invoke *TextField.setTextFormat( )* without specifying any index arguments:

```
this.createTextField("theField_txt", 1, 0, 0, 200, 20);
theField_txt.border = true;
theField_txt.text = "Please enter your name."
// Create a TextFormat object
format = new TextFormat();
format.underline = true;
// Underline the whole field
theField_txt.setTextFormat(format);
```

To underline a single character, apply the format with a single index argument:

```
theField_txt.setTextFormat(0, format); // Underline the first character ("P"
```

To underline a range of characters, apply the format with start and end index arguments:

```
theField_txt.setTextFormat(0, 6, format);  // Underline the characters at in
                                           // 0, 1, 2, 3, 4, and 5 ("Please"
```

The default `underline` value (as contained in the *TextFormat* object returned by *TextField.getTextFormat( )*) for an e text field is `false` ; however, for a completely new *TextFormat* object, it is `null` , indicating that `underline` will n when the format is applied to a text field:

```
trace(anyUnformattedField_txt.getTextFormat().underline);  // Displays: fals
var format = new TextFormat();
trace(format.underline);                                   // Displays: "nul
```

Unlike most web browsers, Flash does not automatically underline hypertext links, so the `underline` property must be indicate links manually, as shown under *`TextFormat` .url* .

See Also

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

TextField .htmlText ,*TextFormat* .url

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

TextFormat.url Property

Flash 6

specifies a hypertext link ***read/write***

theFormat.url

Description

The string `url` property specifies a hypertext link for a span of text.

The *TextFormat*.`url` property has no effect unless *TextField*.`html` is set to `true` for the text field.

To load a document into a named window or frame, supply the window or frame name as the value of *theFormat*.`target`. Flash does not underline or color hypertext links; these features must be inserted manually with the `underline` and `color` properties. For information on formulating links, see the global *getURL( )* function and Appendix E.

To create a link for an entire text field, invoke *TextField.setTextFormat( )* without specifying any index arguments:

```
this.createTextField("theField_txt", 1, 0, 0, 250, 40);
theField_txt.border = true;
theField_txt.text = "Visit www.moock.org.";
theField_txt.html = true;  // REQUIRED!!
// Create a TextFormat object
format = new TextFormat();
format.url = "http://www.moock.org/";
format.underline = true;  // Add the underline manually
// Apply the hypertext link
theField_txt.setTextFormat(format);
```

To add a link to a single character, apply the format with a single index argument:

```
theField_txt.setTextFormat(0, format); // Link the first character, "V"
```

To add a link to a range of characters, apply the format with start and end index arguments:

```
// Link the text: "www.moock.org"
theField_txt.setTextFormat(6, theField_txt.length, format);
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

The default `url` value (as contained in the *TextFormat* object returned by *TextField.getTextFormat( )*) for a text field that contains text is an empty string (no link). For a text field without any text, the `url` value is `null`. For a completely new *TextFormat* object, it is also `null`, indicating that `url` will not be set when the format is applied to a text field:

```
trace(anyUnformattedField_txt.getTextFormat().url);  // Displays nothing
var format = new TextFormat();
trace(format.size);                                  // Displays: "null"
```

See Also

getURL( ), *`TextField`*`.html`, *`TextFormat`*`.color`, *`TextFormat`*`.target`, *`TextFormat`*`.underline`; Appendix E

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Chapter 18. ActionScript Language Reference

toggleHighQuality( ) Global Function Flash 5

change the rendering quality of the PlayerFlash 2; deprecated in

`toggleHighQuality()`

Description

The *toggleHighQuality( )* function switches between High quality and Low quality rendering. When set to High, the Flash Player renders lines with antialiased (smooth) edges. When set to Low, the Flash Player renders lines with aliased (jagged) edges. The *toggleHighQuality( )* function does not take any arguments; it simply switches between the two possible settings—"High" and "Low". This is problematic because it doesn't explicitly set the quality to a known setting, nor does it allow for more than two different possible quality settings.

As of Flash 5, *toggleHighQuality( )* has been deprecated in favor of the global `_quality` property, which supports Low, AutoLow, Medium, High, AutoHigh, and Best rendering settings.

See Also

`_highquality`, `_quality`

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Chapter 18. ActionScript Language Reference

trace( ) Global Function — Flash 4

display a value in the Output window

```
trace(value)
```

Arguments

value

The expression to be resolved and then displayed in the Output window. If the resolved value of `value` is not a string, it is converted to a string before being sent to the Output window, according to the rules described in Table 3-2 in Chapter 3.

Description

The *trace( )* function is a debugging tool used only within the Test Movie mode of the Flash authoring environment. Though unassuming in nature, *trace( )* is actually one of the fundamental components of ActionScript programming; it allows us to check the value of a variable or expression at any point during the playback of a movie. Because the Output window isn't available when we're testing in a browser or the Standalone Player, use an on-screen text field to display results, as follows:

```
this.createTextField("debug_txt", 99999, 50, 50, 300, 200);
debug_txt.border = true;
debug_txt.text += "Your message here" + newline;
```

Alternatively, call a JavaScript alert from the Flash movie, as described at:

http://www.moock.org/asdg/technotes/traceInBrowser

Usage

Multiple *trace( )* statements can slow down movie playback. To view a movie at its genuine speed, turn off tracing under File Publish Settings Flash using the Omit Trace Actions option.

Example

```
trace(firstName);                   // Output the value of firstName
trace(myClip);                      // Output the path to myClip
trace(myClip._x)                    // Output the x coordinate of myClip
trace("hello" + " there");          // Resolve and then output the expressior
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

See Also

Object.watch( ); Chapter 17

About Safari | Terms of Service | Privacy Policy | Contact Us | Help | Submit a Problem

Copyright © 2002 O'Reilly & Associates, Inc. All rights reserved.
1005 Gravenstein Highway North
Sebastopol, CA 95472

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Chapter 18. ActionScript Language Reference

typeof Operator Flash 5

determine the datatype of an expression or identifier

```
typeof operand;
```

Operands

operand

The expression or identifier whose datatype will be determined.

Returns

A string indicating the datatype of the evaluated `operand` as shown in Table 3-4 in Chapter 3.

Description

The *typeof* operator returns the datatype of `operand`. For example, it returns "movieclip" when used with a movie clip identifier.

See Also

instanceof; "The typeof Operator," in Chapter 5; "Determining the Type of an Existing Datum," and Table 3-4 in Chapter 3

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Chapter 18. ActionScript Language Reference

unescape( ) Global Function — Flash 5

decode an escaped (URL-encoded) string

```
unescape(stringExpression)
```

Arguments

stringExpression

A string (or an expression that resolves to a string) that was previously encoded via *escape( )*.

Returns

A new string that is a decoded version of `string`.

Description

The *unescape( )* function returns a new string based on `string`. The new string contains equivalent characters for every occurrence of hexadecimal escape sequences in `string`, as described under *escape( )*. The *escape( )* and *unescape( )* functions are used to encode and decode strings for safe transfer over a network. However, manual use of *unescape( )* rarely is necessary, as Flash automatically unescapes URL-encoded text when it is imported via *loadVariables( )* or the *LoadVars* class.

Example

```
var msg = "hello!";
// Set msgCoded to "hello%21"
msgCoded = escape(msg);
// Set msgDecoded to "hello!"
var msgDecoded = unescape(msgCoded);
```

See Also

escape( ); Appendix B

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Chapter 18. ActionScript Language Reference

unloadMovie( ) Global Function Flash 4

unload a movie or movie clip from the Player

```
unloadMovie(target)
```

Arguments

target

A string or reference indicating the path to the movie clip or document level to unload from the Player (references are converted to paths when used in a string context).

Description

The *unloadMovie( )* function is used most often to unload movies from document levels in the Player. For example, if a movie is loaded onto level 1 in the Player, we can unload that movie from the Player as follows:

```
unloadMovie("_level1");
```

The *unloadMovie( )* function can also be used to unload movie clip instances, in which case it removes the content of the instance but not the instance itself. The instance is left on stage as an empty shell; we can load subsequent *.swf* or *.jpg* files into that shell. Hence, a single clip can act as a container for dynamic content that is managed through successive *loadMovie( )* and *unloadMovie( )* executions. Note that *unloadMovie( )* differs from *removeMovieClip( )* insofar as the latter removes a clip instance from memory, whereas the former removes content from a clip instance but leaves the clip instance shell otherwise intact.

Note that *unloadMovie( )* executes asynchronously—the unload operation does not actually begin until after all scripts on the current frame execute.

Usage

The *unloadMovie( )* function corresponds to the Unload Movie action with a target path in Flash 4.

See Also

loadMovie( ), *MovieClip.removeMovieClip( )*, *MovieClip.unloadMovie( )*, *removeMovieClip( )*, *unloadMovieNum( )*; "Removing Clip Instances and Main Movies," and "Method versus global function overlap issues," in Chapter 13

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Chapter 18. ActionScript Language Reference

unloadMovieNum( ) Global Function Flash 3

remove a movie from a document level

```
unloadMovieNum(level)
```

Arguments

level

A nonnegative integer or an expression that yields a nonnegative integer, indicating the document level to unload.

Description

The *unloadMovieNum( )* function is nearly identical to *unloadMovie( )*, except that it requires the target `level` of the load operation to be specified as a number rather than as a string. This means that *unloadMovieNum( )* can remove movies from document levels only, not from movie clips. Normally, it is used when we wish to dynamically assign the level of a movie to unload, as in:

```
var x = 3;
unloadMovieNum(x);
```

This can also be achieved using string concatenation with the regular *unloadMovie( )* function:

```
unloadMovie("_level" + x);
```

Usage

The *unloadMovieNum( )* function corresponds to the Unload Movie action in Flash 3 and Flash 4, which worked only with numbered levels.

See Also

loadMovieNum( ), MovieClip.removeMovieClip( ), MovieClip.unloadMovie( ), removeMovieClip( ), unloadMovie( ); "Removing Clip Instances and Main Movies," in Chapter 13

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

updateAfterEvent( ) Global Function Flash 5

render the contents of the Stage between frames

```
updateAfterEvent()
```

Description

When a function executes between frames in the Flash Player, any visual changes to the screen are not displayed until the next frame is rendered. However, within the following functions, we can force a screen refresh manually by invoking *updateAfterEvent( )*:

- A *setInterval( )* callback
- *MovieClip.onMouseMove( )*
- *MovieClip.onMouseDown( )*
- *MovieClip.onMouseUp( )*
- *MovieClip.onKeyDown( )*
- *MovieClip.onKeyUp( )*

Note, however, that *updateAfterEvent( )* is not an arbitrary screen-refreshing tool; outside of *setInterval( )* and the handlers in the previous list, it has no effect. In Flash 6, even the *Mouse* and *Key* objects' event handlers do not support *updateAfterEvent( )*; however, they are expected to support it in future versions of the Player.

Example

The following script causes `pointer_mc` to follow the mouse pointer. It uses *updateAfterEvent( )* to refresh the screen every time the pointer moves (which may be more often than the frame rate):

```
pointer_mc.onMouseMove = function () {
  this._x = _root._xmouse;
  this._y = _root._ymouse;
  updateAfterEvent();
}
```

See Also

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

"Refreshing the Screen with updateAfterEvent( )," in Chapter 10

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Chapter 18. ActionScript Language Reference

$version "Global" Property Flash 5

the version of the Flash PlayerFlash 4 Build 11; deprecated in ***read-only***

`_root.$version`

Description

The `$version` property contains the same string as the return value of the global *getVersion( )* function and the `System.capabilities.version` property (the operating system, followed by Player version information). See the `capabilities.version` entry (under "C" in this *Language Reference*) for complete details on the format of the version information and version detection from within a browser before the Flash Player loads.

The `$version` property was introduced midway through the life cycle of Flash Player 4 and was replaced by *getVersion( )* in Flash 5, which itself was replaced by `System.capabilities.version` in Flash 6. Technically, it is not a global property, but a property of the main movie timeline; from any other movie clip timeline, it must be accessed as `_root.$version`.

Usage

Use `$version` only if you are exporting to Flash 4 format; use *getVersion( )* if you are exporting to Flash 5 format; and use `System.capabilities.version` if you are exporting to Flash 6 format.

See Also

`capabilities.version`, *getVersion( )*

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Chapter 18. ActionScript Language Reference

XML Class — **Flash 5; Flash 6 improves performance and adds getBytesLoaded( ) and getBytesTotal( ) methods**

DOM-based support for XML-structured data

Constructor

```
new XML()
new XML(source)
```

Arguments

source

An optional string containing well-formed XML data to be parsed into an XML object hierarchy.

Properties

attributes

An object whose properties store element attributes.

childNodes

An array of references to a node's children.

contentType

The MIME content type to be transmitted to servers.

docTypeDecl

The document's `DOCTYPE` tag.

firstChild

A reference to the first descendant of a node.

ignoreWhite

Determines whether to ignore whitespace nodes during XML parsing.

lastChild

A reference to the last descendant of a node.

loaded

Status of a *load ( )* or *sendAndLoad ( )* operation.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

nextSibling

A reference to the node after this node in the current level of the object hierarchy.

nodeName

The name of the current node.

nodeType

The type of the current node, either 1 (an element node) or 3 (a text node).

nodeValue

The value of the current node.

parentNode

A reference to the immediate ancestor of a node.

previousSibling

A reference to the node before this node in the current level of the object hierarchy.

status

Error code describing the result of parsing XML source into an object hierarchy.

xmlDecl

The document's XML declaration tag.

Methods

appendChild( )

Add a new child node to a node.

cloneNode ( )

Create a copy of a node.

createElement ( )

Create a new element node.

createTextNode ( )

Create a new text node.

getBytesLoaded ( )

The number of downloaded bytes of an external XML file.

getBytesTotal ( )

The physical disk size of an external XML file, in bytes.

hasChildNodes ( )

Check if a node has any descendants.

insertBefore ( )

Add a sibling node before a node.

loa d ( )

Import XML source code from an external document.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

parseXML ( )

Parse a string of XML source code.

removeNode( )

Delete a node from an object hierarchy.

send ( )

Send XML source code to an external script or application.

sendAndLoad ( )

Send XML source code to an external script or application and receive XML source in return.

toString ( )

Convert an *XML* object to a string.

Event Handlers

onData( )

Handler executed when external XML source finishes loading.

onLoad ( )

Handler executed when external XML data has been parsed into an object hierarchy.

Description

We use objects of the *XML* class to manipulate the content of an XML document in an object-oriented manner and to sen XML-formatted data to and from Flash. Using the methods and properties of an *XML* object, we can build an XML-struct document (or read an existing one) and efficiently access, change, or remove the information in that document.

> The *XML* class actually is a subclass of the internal *XMLnode* superclass. See the *XMLnode* class entry for important explanations and clarifications regarding the difference between the two. Table 18-28, also under the *XMLnode* class entry, details which properties and methods are supported for the *XML* and *XMLnode* classes. If the Synopsis for a property or method shows an instance name of `theNode` , either an *XML* or *XMLnode* instance is supported. If the instance name is shown as `xmlDoc` , it must be an instance of the *XML* class and not an instance of the *XMLnode* class. Note that even though the *XML* class is a subclass of the *XMLnode* class, the root node at the top of an XML document hierarchy is an *XML* instance, not an *XMLnode* instance. The subnodes are *XMLnode* instances. Therefore, properties and methods that pertain to the entire XML document and not to individual nodes, such as *load( )* and *send( )* , work with instances of the *XML* class only. There are two types of nodes, *element nodes* and *text nodes* , as described under the *XML* `.nodeType` entry.

The source code of an XML document consists primarily of a series of elements and attributes. For example, in the follow XML fragment, the elements `BOOK` , `TITLE` , `AUTHOR` , and `PUBLISHER` take the same form as well-known HTML t and we see that the `AUTHOR` element supports one attribute, `SALUTATION` :

```
<BOOK>
  <TITLE>ActionScript for Flash MX: The Definitive Guide</TITLE>
  <AUTHOR SALUTATION="Mr.">Colin Moock</AUTHOR>
```

downloaded from: lib.ommolketab.ir

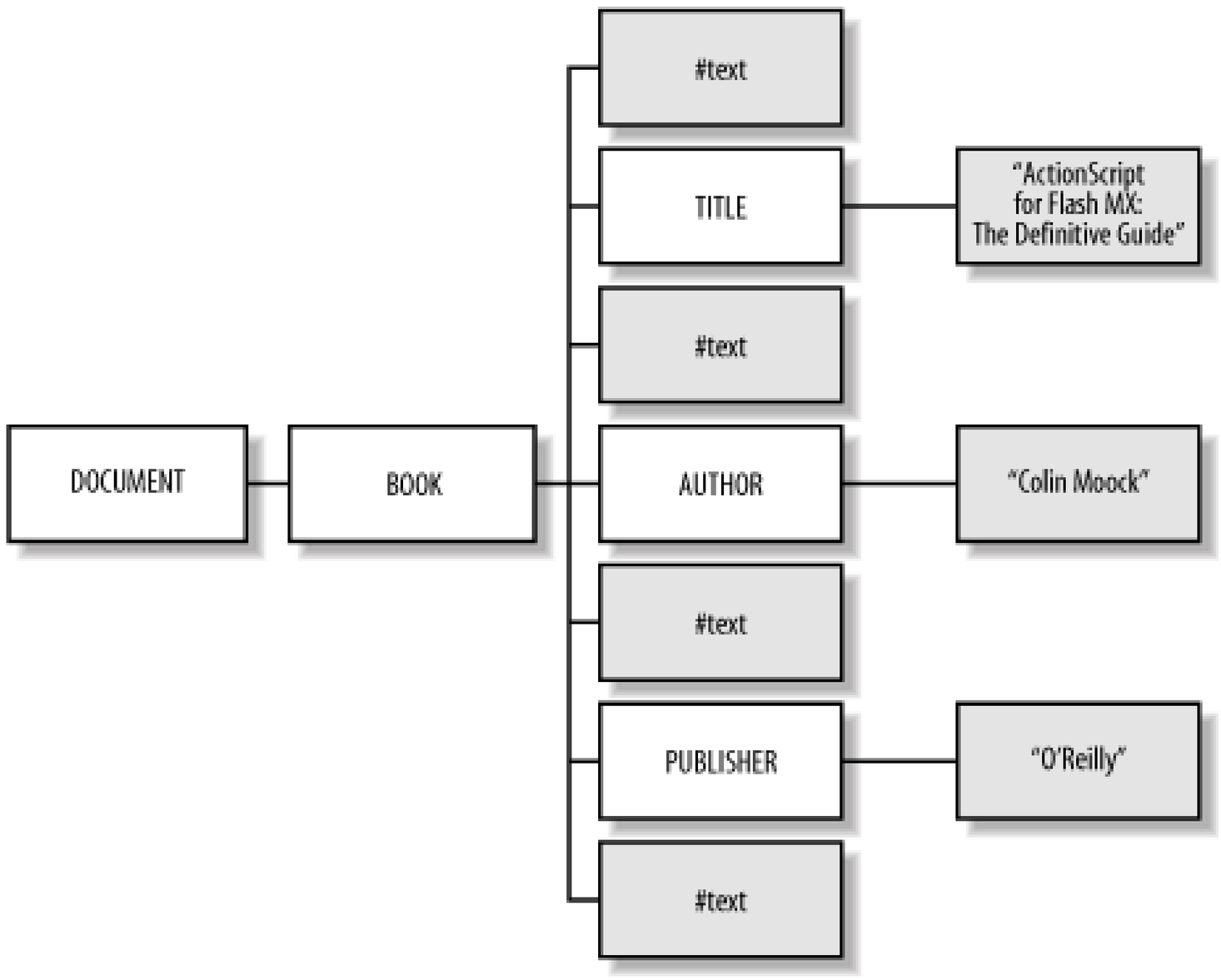
downloaded from: lib.ommolketab.ir

```
  <PUBLISHER>O'Reilly</PUBLISHER>
</BOOK>
```

From an object-oriented perspective, the content of an XML document can be treated as a hierarchy of objects in which ea element and text block becomes an object node in a flowchart-like structure.

Figure 18-7 shows our simple XML `<BOOK>` fragment represented conceptually as an XML object hierarchy.

Figure 18-7. A sample XML node hierarchy



Let's consider the structure and semantics of this sample XML object hierarchy from left to right. We start with the main X object, shown in Figure 18-7 as `DOCUMENT`, which is created automatically by the *XML* constructor and serves as the container for our XML object hierarchy.

Moving one tier to the right in the hierarchy, we come to `BOOK`, which is the first element in our XML source code fragn and, in this case, also the first *object node* under `DOCUMENT`. The `BOOK` node is the *root* of our XML data structure—ev well-formed XML document must have an all-encompassing root element, such as `BOOK`, that contains every other elem (the root of an XML tree has no relation to `_root`, which is a global property signifying the main timeline). Branches o XML object hierarchy are added to the tree either by parsing XML source code or by invoking node-addition methods on objects in the hierarchy.

When a node is contained by another node, the contained node is said to be a *child* of the containing node, which is know the *parent*. In our example, `BOOK` is the first child of `DOCUMENT`, and `DOCUMENT` is `BOOK`'s parent (an XML parent unrelated to the *`MovieClip`*`._parent` property).

As we move to the right in Figure 18-7, we see that `BOOK` has seven children, including four text nodes (shown as #text) do not seem to be present in our original XML document. By default, each occurrence of whitespace between elements in source code is rendered as an object in an XML object hierarchy. If we look closely, we'll find whitespace—a carriage ret

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

and a tab character—between `BOOK` and `TITLE` in the preceding XML fragment. This whitespace is represented by a tex node in Figure 18-7, which also shows similar whitespace nodes after the `TITLE` , `AUTHOR` , and `PUBLISHER` nodes.

`BOOK` 's children are *siblings* of one another (i.e., they reside on the same level in the hierarchy). For example, we say tha `AUTHOR` 's *next sibling* is a text node, and `AUTHOR` 's *previous sibling* is another text node. You can see how the text no get in our way when we're moving from sibling to sibling in a hierarchy. We can deal with these empty whitespace nodes of the following ways:

- By manually stripping them out of our object hierarchy (see the later examples for whitespace-stripping code)
- By detecting and then sidestepping them in our code (see the `nextSibling` and `previousSibling` propert for ways of moving over nodes)
- By simply removing the formatting whitespace in our XML source, ensuring whitespace nodes don't appear in the f place
- By setting our *XML* object's `ignoreWhite` property (available as of Flash 5.0.41.0) to `true` before parsing ou XML source

Finally, we move to the last tier in the hierarchy, where we find that the `TITLE` , `AUTHOR` , and `PUBLISHER` nodes ea have a single child. Each child is a text node, corresponding to the text contained by the elements `TITLE` , `AUTHOR` , an `PUBLISHER` . Notice that the text contained by an element in XML source code resides in a child node of that element i corresponding object hierarchy. To access text contained by an element, we must always refer to that element's child using either `firstChild.nodeValue` or `childNodes[0].nodeValue` , which we'll consider soon.

But what of the element attributes? Where do they appear in our XML object hierarchy? You might expect `AUTHOR` 's `SALUTATION` attribute to be depicted as a child node called `SALUTATION` . But in practice, an attribute is not conside *child* of the element that defines it, but rather a *property* of that element. To learn how attribute properties are accessed, se *XML* `.attributes` entry.

Let's see how to build an XML document as a hierarchy of node objects. To create a new, blank *XML* object, we use the *)* constructor:

```
myDocument = new XML();
```

We can then add nodes to our empty *XML* object by invoking methods such as *appendChild( )* , *parseXML( )* , and *load(* the object. Alternatively, we can create an *XML* object from existing XML source in our script by invoking the XML constructor with the *`source`* argument:

```
myDocument = new XML(source);
```

For example:

```
myDocument = new XML('<P>hello world!</P>');
```

When a *`source`* argument is supplied to the *XML( )* constructor, *`source`* is parsed and converted into a new object hierarchy, which is then stored in the object returned by the constructor. (In this case, the node `P` is assigned as `myDocur` 's first child and the text node with the `nodeValue` "hello world!" is assigned as `P` 's first child.)

Once an XML hierarchy is created and stored in an object, we can access the information in that hierarchy using the meth and properties of the *XML* class. For example, suppose we want to retrieve the text "hello world!" in `myDocument` . Thinking in object-oriented terms, we might assume that we can access the text of `P` as a property of `myDocument` , as follows: `myDocument.P` . In fact, that won't work; instead of referring to nodes by name, we use the *XML* class's built

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

properties, such as `firstChild` and `childNodes` , to access nodes. For example, to access the `P` node, we can use:

```
myDocument.firstChild       // Accesses P
myDocument.childNodes[0]    // Also accesses P
```

Because `firstChild` returns a reference to the first child node of the specified node in the hierarchy, `myDocument.firstChild` returns a reference to node `P` . However, we want the text "hello world!" contained by `P` node `P` itself. As we saw earlier, the text of an element node is stored within a child of that node. Therefore, we can refere the text node (i.e., the first descendant of `P`), like this:

```
myDocument.firstChild.firstChild  // Accesses the text node under P
```

To obtain the value of a node, we use the `nodeValue` property. For example, we can display the value "hello world!" ir Output window using:

```
trace(myDocument.firstChild.firstChild.nodeValue);
```

Or, we can reassign the value of the text node under `P` using:

```
myDocument.firstChild.firstChild.nodeValue = "goodbye cruel world";
```

To perform other node-related operations, such as removing the `P` node altogether, adding a new node, or moving the text "hello world!" to another node, we invoke appropriate methods of the *XML* class. For example:

```
// Delete P
myDocument.firstChild.removeNode();

// Make a new element named P
newElement = myDocument.createElement("P");

// Add the new element to our document
myDocument.appendChild(newElement);

// Make a new text node to attach to P
newText = myDocument.createTextNode("XML is fun");

// Attach the new text node to P
myDocument.firstChild.appendChild(newText);
```

As you can see, working with XML-structured data in an object hierarchy is a mediated endeavor. We build, destroy, and manipulate the data by invoking methods on, and accessing properties of, objects. To learn the various tools available for working with XML data, explore the properties and methods of the *XML* class, which are listed after this entry.

In Flash Player 6, XML text should be encoded using the Unicode UTF-8, UTF-16BE, or UTF-16LE formats, which allov the inclusion of international characters. When a file is UTF-16 encoded, it is expected to start with a *byte order marker* (BOM) indicating whether the encoding is big-endian or little-endian. Most text editors add the BOM automatically. Whe BOM is present in a file, the encoding is assumed to be UTF-8. When in doubt, you should use UTF-8 encoding, where by order is not an issue. For more information on UTFs and BOMs, see:

http://www.unicode.org/unicode/faq/utf_bom.html

ActionScript manipulates XML data using the Document Object Model (DOM) standard published by the World Wide W Consortium (W3C). For thorough coverage of how the DOM represents XML-structured data as an object hierarchy, cons

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

http://www.w3.org/DOM

For details on the language-independent specifications of the core DOM, see:

http://www.w3.org/TR/2000/REC-DOM-Level-2-Core-20001113/core.html

Pay particular attention to "Interface Node" under 1.2, *Fundamental Interfaces* .

For details on how the DOM is implemented in ECMA-262, see:

http://www.w3.org/TR/2000/REC-DOM-Level-2-Core-20001113/ecma-script-binding.html

Usage

Note that all custom subclasses of the *XML* class that you write must invoke *super( )* from their constructor function. This ensures that the methods and properties of the *XML* class are available to instances of the subclass.

For example:

```
function XMLsub (src) {
  // Required! Pass raw XML source string to XML( ) constructor.
  super(src);
}

XMLsub.prototype = new XML();

theDoc = new XMLsub();
theDoc.parseXML("<TEST>hello world</TEST>");
```

Example

We've seen that the whitespace between any two elements in XML source code is represented by a text node in the corresponding XML object hierarchy. Prior to Flash Player 5.0.41.0, undesired whitespace nodes had to be stripped out of XML object hierarchy manually. Stripping a particular kind of node is a common task in XML handling and serves as a g example of *tree traversal* (moving through every node in a hierarchy). Let's consider two different techniques for stripping whitespace nodes from a document.

In the first example, we'll use a classic FIFO (First In First Out) stack to add all the nodes in a tree to an array for processi The *stripWhitespaceTraverse( )* function seeds an array of node elements with `theNode` , which it receives as an argun Then it enters a loop in which it removes the first node in the array, processes that node, and adds its children (if any) to th array. When the array has no more elements, all the descendants of `theNode` have been processed. During processing, node that has no children is considered potential whitespace (because text nodes never have children). Each of these nodes checked to see if:

- It is a text node (as determined by its `nodeType` property).
- It contains any characters above ASCII 32, which are not considered whitespace.

Any text nodes containing characters below ASCII 32 only (i.e., only whitespace) are removed:

```
// Strip Whitespace Using a FIFO Stack
// Function: stripWhitespaceTraverse()
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

```
//  Version: 1.0.1
//     Desc: Strips whitespace nodes from an XML document by traversing the
function stripWhitespaceTraverse (theNode) {
  // Create a list of nodes to process.
  var nodeList = new Array();
  // Seed the list with the node passed to the function.
  nodeList[0] = theNode;

  // Create a local variable to track whether we've found an empty node.
  var emptyNode;
  // Local counter variable.
  var i = 0;

  // Process the passed node and its descendents until there are none left.
  while (nodeList.length > 0) {
    // Grab the first node on the list, and remove it from the list.
    currentNode = nodeList.shift();

// Assume we're dealing with a nonempty node.
emptyNode = false;

// If this node has children...
if (currentNode.childNodes.length > 0) {
  // ...add this node's children to the list of nodes to process.
  nodeList = nodeList.concat(currentNode.childNodes);
} else {
  // ...otherwise, this node is the end of a branch, so check if it's a
  // text node...
  if (currentNode.nodeType =  = 3) {
    // Yup, it's a text node, so check if it contains empty whitespace o
    // Assume it's only whitespace and try to prove the assumption wrong
    emptyNode = true;
    for (i = 0; i < currentNode.nodeValue.length; i++) {
      // A useful character is anything over 32 (space,
      // tab, new line, etc. are all below 32).
      if (currentNode.nodeValue.charCodeAt(i) > 32) {
        emptyNode = false;
        break;
      }
    }
  }

  // If no useful characters were found, delete the node
  if (emptyNode) {
    currentNode.removeNode();
  }
 }
 }
}
```

Traditionally, the technique shown in this example is very efficient. However, in Flash Player 5, the *Array.concat( )* metl executes quite slowly. Hence, in Flash 5 it's quicker to strip whitespace using the technique shown in the following examp

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Study the comments carefully:

```
// Strip Whitespace Using Function Recursion
// Strips whitespace nodes from an XML document
// by passing twice through each level in the tree.
function stripWhitespaceDoublePass (theNode) {
  // Loop through all the children of theNode.
  for (var i = 0; i < theNode.childNodes.length; i++) {
    // If the current node is a text node...
    if (theNode.childNodes[i].nodeType =  = 3) {

      // ...check for any useful characters in the node.
      var j = 0;
      var emptyNode = true;
      for (j = 0;j < theNode.childNodes[i].nodeValue.length; j++) {
        // A useful character is anything over 32 (space, tab,
        // new line, etc. are all below 32).
        if (theNode.childNodes[i].nodeValue.charCodeAt(j) > 32) {
          emptyNode = false;
          break;
        }
      }

      // If no useful characters were found, delete the node.
      if (emptyNode) {
        theNode.childNodes[i].removeNode();
      }
    }
  }

  // Now that all the whitespace nodes have been removed from theNode,
  // call stripWhitespaceDoublePass( ) recursively on its remaining children
  for (var k = 0; k < theNode.childNodes.length; k++) {
    stripWhitespaceDoublePass(theNode.childNodes[k]);
  }
}
```

See Also

The *XMLnode* class, the *XMLSocket* class; Appendix E

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

XML.appendChild( ) Method — Flash 5

add a new child node to a node, or move an existing node

```
theNode.appendChild(childNode)
```

Arguments

childNode

An existing XML node object.

Description

The *appendChild( )* method adds the specified `childNode` to `theNode` as `theNode` 's last child, where `theNode` can be an *XML* or *XMLnode* instance. We can use *appendChild( )* to add a new node to an existing node, to move a node within a document, or to move a node between documents. In each of these cases, `childNode` must be a reference to a node object that already exists.

To add a new child node to an existing node, we must first create the new child node using *createElement( )* , *createTextNode( )* , or *cloneNode( )* or by parsing XML source code into an *XML* object. For example, in the following code, we create a new `P` node and a new text node. We append the text node to the `P` node and then append the `P` node and its text node child to the top node of a document:

```
// Create a document
myDoc = new XML('<P>paragraph 1</P>');

// Create a P node and a text node
newP = myDoc.createElement("P");
newText = myDoc.createTextNode("paragraph 2");

// Append the text node to the P node
newP.appendChild(newText);

// Append the P node (including its text child) to myDoc
myDoc.appendChild(newP);

trace(myDoc);  // Displays: "<P>paragraph 1</P><P>paragraph 2</P>"
```

To move a node within a document, specify a `childNode` that is a reference to an existing node in the document. In this situation, `childNode` indicates the old location of the node and `theNode` indicates the new parent of the node. In the process of being appended to `theNode` , `childNode` is removed from its previous parent node. For

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

example, here we move the B node from its parent P node to the root of the document:

```
// Create a new document
myDoc = new XML('<P>paragraph 1<B>bold text</B></P>');

// Store a reference to the B node
boldText = myDoc.firstChild.childNodes[1];

// Append the B node to the root of the document, while removing it from P
myDoc.appendChild(boldText);

trace(myDoc);  // Displays: "<P>paragraph 1</P><B>bold text</B>"
```

We also could have skipped the reference-storing step and just moved the node directly:

```
myDoc.appendChild(myDoc.firstChild.childNodes[1]);
```

To move a node between documents, *childNode* should be a reference to a node in the first (source) document and *theNode* should be a reference to a node in the second (target) document. For example, here we move the B node from myDoc to myOtherDoc :

```
myDoc = new XML('<P>paragraph 1<B>bold text</B></P>');
myOtherDoc = new XML();

myOtherDoc.appendChild(myDoc.firstChild.childNodes[1]);

trace(myDoc);       // Displays: "<P>paragraph 1</P>"
trace(myOtherDoc);  // Displays: "<B>bold text</B>"
```

See Also

XML.cloneNode( ) , XML.createElement( ) , XML.createTextNode( ) , XML.insertBefore( ) , XML.removeNode( )

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

XML.attributes Property

Flash 5

an object whose properties store element attributes

read/write

```
theNode.attributes.attributeIdentifier
theNode.attributes[attributeNameInQuotes]
```

Description

The `attributes` property stores the names and values of attributes defined by *`theNode`*, where *`theNode`* can be an *XML* or *XMLnode* instance. For example, the `ALIGN` attribute of this `P` tag:

```
<P ALIGN="CENTER">this is a paragraph</P>
```

is accessed using *`theNode`*`.attributes.ALIGN` or *`theNode`*`.attributes["ALIGN"]`. If the `P` tag is the only tag in our XML source, we can access the `ALIGN` attribute as follows:

```
// Create an XML object hierarchy
myDoc = new XML('<P ALIGN="CENTER">this is a paragraph</P>');

// Access the ALIGN attribute. Displays: "CENTER"
trace(myDoc.firstChild.attributes.ALIGN);

// Set the ALIGN attribute
myDoc.firstChild.attributes.ALIGN = "LEFT";
```

The `attributes` property is itself an object. We can add new properties to the `attributes` object, thereby adding new attributes to *`theNode`*, as follows:

```
// Add a CLASS attribute to the P tag
myDoc.firstChild.attributes.CLASS = "INTRO";

// firstChild now represents the XML source:
// <P ALIGN="CENTER" CLASS="INTRO">this is a paragraph</P>
```

Because `attributes` is not an array, it doesn't contain a `length` property to indicate the number of properties/attributes it includes. Instead, we can access all the attributes defined on an element using a *for-in* loop:

```
var count = 0;
for (var prop in theNode.attributes) {
  trace("attribute " + prop + " has the value " + theNode.attributes[prop]);
  count++;
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

```
}
trace("The node has " + count + " attributes.");
```

If the XML element represented by *theNode* has no attributes, `attributes` is an empty object with no properties, and the preceding example would indicate zero attributes.

See Also

XML `.nodeType`

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Chapter 18. ActionScript Language Reference

XML.childNodes Property — Flash 5

an array of references to a node's children — ***read-only***

theNode.childNodes[*n*]

Description

The `childNodes` property is an array whose elements contain references to the immediate descendants of *`theNode`* where *`theNode`* can be an *XML* or *XMLnode* instance. The `childNodes` property is used to access nodes in an XML hierarchy. For example, if we create an object hierarchy as follows:

```
myDoc = new XML('<STUDENT><NAME>Tim</NAME><MAJOR>BIOLOGY</MAJOR></STUDENT>')
```

we can then access the `STUDENT` node using:

```
myDoc.childNodes[0];
```

We can access the `NAME` and `MAJOR` nodes (which are descendants of `STUDENT`) using:

```
myDoc.childNodes[0].childNodes[0];     // NAME
myDoc.childNodes[0].childNodes[1];     // MAJOR
```

If the hierarchy below *`theNode`* changes, `childNodes` is automatically updated to reflect the new structure. For example, if we delete the `MAJOR` node, `myDoc.childNodes[0].childNodes[1]` will return `undefined` .

We often refer to nodes to manipulate information or rearrange a document's structure. For example, we can change a student's name or add a new student using:

```
// Check the name of the student
trace("The student's name is: "
      + myDoc.childNodes[0].childNodes[0].childNodes[0].nodeValue);

// Change the name of the student
myDoc.childNodes[0].childNodes[0].childNodes[0].nodeValue = "James";

// Copy the STUDENT node
newStudent = myDoc.childNodes[0].cloneNode(true);

// Add a new STUDENT node to the document
myDoc.appendChild(newStudent);
```

Note that, as a convenience, we can also use the `firstChild` property to refer to `childNodes[0]` . The following

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

references are identical:

```
myDoc.childNodes[0];
myDoc.firstChild;
```

To iterate through all the children of a node, we can use a *for* statement, as follows:

```
for (var i = 0; i < theNode.childNodes.length; i++) {
  trace("child " + i + " is " + theNode.childNodes[i].nodeName);
}
```

However, our example traverses only the first level of *theNode* 's hierarchy. The examples under the *XML* .nextSibling entry show how to access all the nodes below *theNode* . If *theNode* has no children, *theNode* .childNodes.length is 0.

Usage

Remember that empty text nodes, representing the whitespace used to format XML source code, also show up in a childNode list. For example, in the following XML source, empty text nodes are created by the whitespace after the BOOK start tag and the TITLE , AUTHOR , and PUBLISHER end tags:

```
<BOOK>
  <TITLE>ActionScript for Flash MX: The Definitive Guide</TITLE>
  <AUTHOR SALUTATION="Mr.">Colin Moock</AUTHOR>
  <PUBLISHER>O'Reilly</PUBLISHER>
</BOOK>
```

Hence, the first child node of BOOK is an empty text node; the second child is TITLE . To remove whitespace, use *XML* .ignoreWhite .

Bugs

In Flash Player 5, an XML document with an XML declaration (<?xml version="1.0"?>) has a childNodes[0] of undefined . The first child of the document is stored erroneously in childNodes[1] . This bug was fixed in Flash Player 6.

See Also

XML .firstChild , *XML.hasChildNodes( )* , *XML* .lastChild , *XML* .nextSibling , *XML* .previousSibling

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

XML.cloneNode( ) Method — Flash 5

create an independent copy of a node

```
theNode.cloneNode(deep)
```

Arguments

deep

A Boolean indicating whether to recursively include `theNode`'s children in the clone operation. If `true`, clone the entire hierarchy starting at `theNode`. If `false`, clone only `theNode` itself (and its attributes, if it is an element node).

Returns

A duplicate of the `theNode` object, optionally including its subtree.

Description

The *cloneNode( )* method creates and returns a copy of `theNode`, where `theNode` can be an *XML* or *XMLnode* instance. If `theNode` is an element node, the clone includes all of `theNode`'s attributes and values. If `deep` is `true`, the returned copy includes the entire node hierarchy descending from `theNode`.

We often use *cloneNode( )* to create a new, independent node based on an existing template (which saves us from generating the new node structure manually). Normally, once we've cloned a node, we customize it and insert it into an existing XML document using either *appendChild( )* or *insertBefore( )*. The following example clones the first paragraph of a document to make a sibling paragraph with the same structure:

```
// Create a new document
myDoc = new XML('<P>paragraph 1</P>');

// Make a clone of the first paragraph
newP = myDoc.firstChild.cloneNode(true);

// Customize the clone
newP.firstChild.nodeValue = "paragraph 2";

// Add the clone into the document
myDoc.appendChild(newP);

trace(myDoc);  // Displays: "<P>paragraph 1</P><P>paragraph 2</P>"
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Note that the text in an element is stored in a separate child node of that element, so we must set `deep` to `true` to preserve an element's text content in a clone operation. Remember that *cloneNode( )* does not insert the element it returns into the node's hierarchy—we must do that ourselves using *appendChild( )* or *insertBefore( )*.

See Also

XML.appendChild( ), *XML.createElement( )*, *XML.createTextNode( )*, *XML.insertBefore( )*

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Chapter 18. ActionScript Language Reference

XML.contentType Property — Flash 5.0.41.0

MIME content type for XML data sent via XML.send( ) and XML.sendAndLoad( ) — *read/write*

`xmlDoc.contentTypeAccess`

Description

The `contentType` property is the MIME type that is sent to a server when *XML.send( )* or *XML.sendAndLoad( )* is invoked. It defaults to `"application/x-www-form-urlencoded"`, but that MIME type is used only cosmetically to encourage interoperability with server-side applications; the data sent is not URL-encoded. The specified *`xmlDoc`* must be the top-level node in an XML object hierarchy (i.e., an instance of the *XML* class, not the *XMLnode* class). When set on an individual *XML* instance, the `contentType` property affects only that instance. For example, the following code sets the MIME for `doc` only:

```
doc.contentType = "application/xml";
```

To affect all *XML* objects, set the value of `XML.prototype.contentType`. For example, the following code sets the MIME of all XML objects in a movie to `"application/xml"`:

```
XML.prototype.contentType = "application/xml";
```

The `contentType` property first appeared in Flash Player 5.0.41.0, before which it was not possible to set the MIME type. Check `contentType`'s validity in the Player by comparing it to `undefined`, or use the *getVersion( )* function to determine the Player version and build.

See Also

XML.send( ), XML.sendAndLoad( )

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

XML.createElement( ) Method — Flash 5

create a new element node

```
xmlDoc.createElement(tagName)
```

Arguments

tagName

A case-sensitive string indicating the name of the element to create. For example, in the tag, `<P ALIGN="RIGHT">`, `P` is the tag name.

Returns

A new element node object, with no parent and no children.

Description

The *createElement( )* method is our primary means of generating new element nodes for inclusion in an XML document object hierarchy. The specified `xmlDoc` must be the top-level node in an XML object hierarchy (i.e., an instance of the *XML* class, not the *XMLnode* class). Note that *createElement( )* does not insert the element it returns into `xmlDoc`—we must do that ourselves using *appendChild( )* or *insertBefore( )*. For example, here we create and insert a new `P` element into a document:

```
myDoc = new XML();
newP = myDoc.createElement("P");
myDoc.appendChild(newP);
```

We can combine those steps by using this:

```
myDoc.appendChild(myDoc.createElement("P"));
```

The *createElement( )* method cannot be used to create text nodes; use *createTextNode( )* instead.

See Also

XML.appendChild( ), XML.cloneNode( ), XML.createTextNode( ), XML.insertBefore( )

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

XML.createTextNode( ) Method — Flash 5

create a new text node

```
xmlDoc.createTextNode(text)
```

Arguments

text

A string containing the text that is to become the `nodeValue` of the new node.

Returns

A new text node object, with no parent and no children.

Description

The *createTextNode( )* method is our primary means of generating new text nodes for inclusion in an XML document object hierarchy. The specified `xmlDoc` must be the top-level node in an XML object hierarchy (i.e., an instance of the *XML* class, not the *XMLnode* class). Note that *createTextNode( )* does not insert the element it returns into `xmlDoc`—we must do that ourselves using *appendChild( )* or *insertBefore( )*. For example, here we create and insert a new `P` element into a document, and then we give that `P` element a text-node child:

```
myDoc = new XML();
newP = myDoc.createElement("P");
myDoc.appendChild(newP);

newText = myDoc.createTextNode("This is the first paragraph");
myDoc.firstChild.appendChild(newText);

trace(myDoc);  // Displays: "<P>This is the first paragraph</P>"
```

Normally, text nodes are stored as the children of element nodes, which are created using *createElement( )*.

The *createTextNode( )* method converts the characters <, >, &, ', and " according to the rules of entity conversion discussed under *XML.parseXML( )*.

See Also

XML.appendChild( ), *XML.cloneNode( )*, *XML.createElement( )*, *XML.insertBefore( )*, *XML.parseXML( )*

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Chapter 18. ActionScript Language Reference

XML.docTypeDecl Property — Flash 5

the document's DOCTYPE tag — *read/write*

xmlDoc.docTypeDecl

Description

The docTypeDecl string property specifies the DOCTYPE tag of *xmlDoc*, if any exists. Otherwise, docTypeDecl is undefined. The specified *xmlDoc* must be the top-level node in an XML object hierarchy (i.e., an instance of the *XML* class, not the *XMLnode* class).

An XML document's DOCTYPE specifies the name and location of the *document type declaration* (DTD) used to validate the document. ActionScript does not perform validation of XML documents; it merely parses them. We use the DOCTYPE tag to build XML documents that may be validated externally or to identify the type of a loaded XML document.

Example

```
var myXML = new XML('<?xml version="1.0"?><!DOCTYPE foo SYSTEM "bar.dtd">'
                    + '<P>a short document</P>');

trace(myXML.docTypeDecl);  // Displays: "<!DOCTYPE foo SYSTEM "bar.dtd">"

// Set a new DOCTYPE
myXML.docTypeDecl = '<!DOCTYPE baz SYSTEM "bam.dtd">';
```

See Also

XML .xmlDecl

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Chapter 18. ActionScript Language Reference

XML.firstChild Property — Flash 5

a reference to the first descendant of a node — ***read-only***

theNode.firstChild

Description

The `firstChild` property is synonymous with `childNodes[0]`. It returns a reference to the first node object that descends from *`theNode`*, where *`theNode`* can be an *XML* or *XMLnode* instance. If *`theNode`* has no children, `firstChild` returns `null`.

In this XML source fragment, the `firstChild` of the `MESSAGE` node is the text node with the `nodeValue` "hey":

```
<!-- Fragment 1 -->
<MESSAGE>hey</MESSAGE>
```

Here, the `firstChild` of the `HOTEL` node is the `ROOM` node:

```
<!-- Fragment 2 -->
<HOTEL><ROOM><SIZE>Double</SIZE></ROOM></HOTEL>
```

When *`theNode`* is the top of the object hierarchy (i.e., when it is an instance of the *XML* class, which refers to an *XML* document object and not an instance of the *XMLnode* class), `firstChild` may not always be a reference to the first useful element in the document. If a document includes an XML declaration (`<?xml version="1.0"?>`) and perhaps a `DOCTYPE` tag, normally there are whitespace nodes before the actual root element of the XML hierarchy (to remove them, use *`XML`* `.ignoreWhite`). If an XML fragment has no XML declaration, no `DOCTYPE`, and no whitespace, we can start processing it with the document's `firstChild` node, as in:

```
// Create a new XML fragment
myDoc = new XML('<MESSAGE><USER>gray</USER><CONTENT>hi</CONTENT></MESSAGE>')

// Store the XML fragment's first node in the variable msg
msg = myDoc.firstChild;

// Assign the text contained by the USER tag
// to a text field called userNameOutput
userNameOutput = msg.firstChild.firstChild.nodeValue;
```

It's good form to use `nodeValue` to access the text contained by the `USER` tag. When we use a text-node object in a string context, the *toString( )* method is automatically invoked on that node, and the text in the node is returned.

See Also

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

XML .childNodes,*XML* .lastChild,*XML* .nextSibling,*XML* .previousSibling

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Chapter 18. ActionScript Language Reference

XML.getBytesLoaded( ) Method Flash 6

the number of downloaded bytes of an external XML file

```
xmlDoc.getBytesLoaded()
```

Returns

An integer representing the number of bytes that have finished downloading to the Player. Divide by 1024 to convert bytes to kilobytes (KB).

Description

The *getBytesLoaded( )* method tells us the number of bytes of an external XML file that have downloaded into the Flash Player due to an earlier call to *load( )* or *sendAndLoad( )*. The specified `xmlDoc` must be the top-level node in an XML object hierarchy (i.e., an instance of the *XML* class, not of the *XMLnode* class).

For a new *XML* object, *getBytesLoaded( )* returns `undefined`. Immediately after *load( )* or *sendAndLoad( )* is called, *getBytesLoaded( )* returns 0. As data arrives, *getBytesLoaded( )* returns the number of bytes transferred so far, until the entire XML file has loaded. From then on, it returns the total number of bytes last transferred, until the next call to *load( )* or *sendAndLoad( )*.

The *getBytesLoaded( )* and *getBytesTotal( )* methods are used together to determine (and, normally, display) the load progress of a downloading XML file. However, merely detecting the completion of a load operation does not require *getBytesLoaded( )*; the *onLoad( )* event handler is automatically executed when an XML file finishes loading. The success of the load attempt should be checked from the *onLoad( )* handler.

Example

The following code, available at the online Code Depot, adds two new methods to all *XML* document objects: *preload( )* and *sendAndPreload( )*. They function the same as *load( )* and *sendAndLoad( )*, except that they use *getBytesLoaded( )* to report the progress while the XML file is loading. Any *XML* object that invokes *preload( )* or *sendAndPreload( )* is expected to define a custom *onBytesLoaded( )* callback that will be invoked every 200 milliseconds and supplied with load progress information (bytes loaded, percent complete, etc.). Typically, the code in *onBytesLoaded( )* displays the load progress on screen. The *XML* object should also define an *onLoad( )* handler to respond to completion of the load operation. When used with *preload( )* or *sendAndPreload( )*, the *onLoad( )* function has the added responsibility of canceling any failed preload attempts, as shown in the following example:

```
/*
 * Method: XML.checkLoadProgress()
 *   Desc: Checks progress of a load() or sendAndLoad() operation.
 *         Called only by preload() or sendAndPreload(). Reports progress
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

```
 *          by invoking a custom XML.onBytesLoaded() handler.
 */
XML.prototype.checkLoadProgress = function () {
  // Check how many KB have loaded.
  var kbLoaded = Math.floor(this.getBytesLoaded()/1024);
  // Check the size of the file loading, in KB.
  var kbTotal = Math.floor(this.getBytesTotal()/1024);
  // Calculate the percent complete.
  var percentDone = isNaN(Math.floor(kbLoaded/kbTotal * 100)) ?
                    0 : Math.floor(kbLoaded/kbTotal * 100);
  // Execute onBytesLoaded(), which typically would display load progress.
  this.onBytesLoaded(this.getBytesLoaded(),
                     this.getBytesTotal(),
                     kbLoaded,
                     kbTotal,
                     percentDone);
  // If the amount loaded is equal to the total file size,
  // the file is finished loading. However, we must ensure
  // that getBytesLoaded() returns a value greater than 0
  // because both getBytesLoaded() and getBytesTotal()
  // return 0 until the first bit of data arrives.
  if (this.getBytesLoaded() > 5
      && this.getBytesLoaded() =  = this.getBytesTotal()) {
    // ...stop calling checkLoadProgress().
    this.clearLoadCheck();
  }
}

/*
 * Method: XML.clearLoadCheck()
 *   Desc: Cancels an interval calling checkLoadProgress().
 */
XML.prototype.clearLoadCheck = function () {
  // If there is a load interval for this XML object...
  if (this.loaderID) {
    // ...stop calling it.
    clearInterval(this.loaderID);
  }
}

/*
 * Method: XML.preload()
 *   Desc: Begins the download of an XML file and executes
 *         checkLoadProgress() every 200 ms until the file has loaded.
 */
XML.prototype.preload = function (url) {
  // Always stop any previous preload before proceeding.
  this.clearLoadCheck();

  // Load the XML file.
  this.load(url);
  // Call checkLoadProgress() every 200 ms.
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

```
  this.loaderID = setInterval(this, "checkLoadProgress", 200);
}

/*
 * Method: XML.sendAndPreload()
 *   Desc: Sends XML to the server, and downloads the response. Executes
 *         checkLoadProgress() every 200 ms until the file has loaded.
 */
XML.prototype.sendAndPreload = function (url, resultXML) {
  // Always stop any previous preload before proceeding.
  this.clearLoadCheck();

  // If the target XML object is valid...
  if (resultXML instanceof XML) {
    // ...send and load the XML file.
    this.sendAndLoad(url, resultXML);
    // Call checkLoadProgress() every 200 ms.
    this.loaderID = setInterval(this, "checkLoadProgress", 200);
  }  else {
    // ...otherwise quit without performing the load.
    return false;
  }
}

// Usage:
// Create an XML object.
doc = new XML();
// Assign an onLoad() callback.
doc.onLoad = function (success) {
  // If the load was successful...
  if (success) {
    // ...process the new XML.
    trace("done loading!");
  } else {
    // ...otherwise, perform failure operations and
    // cancel the preload interval (REQUIRED!).
    trace("load failed");
    this.clearLoadCheck();
  }
}
// Assign an onBytesLoaded() callback. This is where each XML document
// can decide what to do with the preload information.
doc.onBytesLoaded = function (bytesLoaded, bytesTotal,
                              kbLoaded, kbTotal, percentLoaded) {
  trace("Loading: " + kbLoaded +  " of " + kbTotal);
  trace(percentLoaded + " percent complete.");
}
// Preload the XML file.
doc.preload("http://www.yoursite.org/datafiles/someInfo.xml");
```

See Also

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

setInterval( ), XML.getBytesTotal( ), XML.load( ), XML.onLoad( ), XML.sendAndLoad( )

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Chapter 18. ActionScript Language Reference

XML.getBytesTotal( ) Method Flash 6

the physical disk size of an external XML file, in bytes

```
xmlDoc.getBytesTotal()
```

Returns

An integer representing the size of the XML file, in bytes. Divide by 1024 to convert bytes to kilobytes (KB).

Description

The *getBytesTotal( )* method tells us the byte size of an external file being loaded into `xmlDoc` via *XML.load( )* or *XML.sendAndLoad( )*. The specified `xmlDoc` must be the top-level node in an XML object hierarchy (i.e., an instance of the *XML* class, not the *XMLnode* class).

For a new *XML* object, *getBytesTotal( )* returns `undefined`. Immediately after *load( )* or *sendAndLoad( )* is called, *getBytesTotal( )* returns 0 until the size of the XML file being loaded is determined. It then returns the total byte size of the loading file until the next call to *load( )* or *sendAndLoad( )*.

The *getBytesTotal( )* and *getBytesLoaded( )* methods are used together to determine (and, normally, display) the load progress of a downloading XML file. See *getBytesLoaded( )* for details and example code.

See Also

XML.getBytesLoaded( ), *XML.onLoad( )*, *XML.load( )*, *XML.sendAndLoad( )*

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

XML.hasChildNodes( ) Method Flash 5

check if a node has any descendants

```
theNode.hasChildNodes()
```

Returns

A Boolean: `true` if *`theNode`* has any children, `false` if it does not.

Description

The *hasChildNodes( )* method indicates whether any node hierarchy extends from *`theNode`*, where *`theNode`* can be an *XML* or *XMLnode* instance. The *hasChildNodes( )* method is synonymous with the comparison expression:

```
theNode.childNodes.length > 0
```

If *`theNode`* contains no subnodes, *hasChildNodes( )* returns `false`.

Example

We can use *hasChildNodes( )* to determine whether to operate on a node during node processing. For example, here we remove the nodes below the first child of a document until the first child has no more children:

```
while (myDoc.firstChild.hasChildNodes()) {
  myDoc.firstChild.firstChild.removeNode();
}
```

See Also

XML.childNodes

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

About Safari | Terms of Service | Privacy Policy | Contact Us | Help | Submit a Problem

Copyright © 2002 O'Reilly & Associates, Inc. All rights reserved.

1005 Gravenstein Highway North
Sebastopol, CA 95472

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Chapter 18. ActionScript Language Reference

XML.ignoreWhite Property — Flash 5.0.41.0

determines whether to ignore whitespace nodes during XML parsing — ***read/write***

xmlDoc.ignoreWhite

Description

The Boolean `ignoreWhite` property dictates whether to discard text nodes containing only whitespace during the parsing process. The default value is `false` (don't throw away whitespace nodes). This is an instance-wide setting that applies to an entire XML document, not just to a specific node. Therefore, the specified *`xmlDoc`* must be the top-level node in an XML object hierarchy (i.e., an instance of the *XML* class, not the *XMLnode* class).

Example

To cause a single XML document to discard whitespace nodes during parsing, use:

```
myXML.ignoreWhite = true;
```

To cause all XML documents to discard whitespace nodes during parsing, use:

```
XML.prototype.ignoreWhite = true;
```

The `ignoreWhite` property must be set before XML parsing occurs (typically due to a *load( )* or *sendAndLoad( )* operation).

See Also

The manual whitespace-stripping code examples under the *XML* class entry

About Safari | Terms of Service | Privacy Policy | Contact Us | Help | Submit a Problem

Copyright © 2002 O'Reilly & Associates, Inc. All rights reserved.
1005 Gravenstein Highway North
Sebastopol, CA 95472

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Chapter 18. ActionScript Language Reference

XML.insertBefore( ) Method — Flash 5

insert a new sibling node before an existing node

```
theNode.insertBefore(newChild, beforeChild)
```

Arguments

newChild

An existing XML node object.

beforeChild

The child of `theNode` before which `newChild` should be inserted.

Description

The *insertBefore( )* method adds `newChild` to `theNode` 's child list, before `beforeChild` (`theNode` can be *XML* or *XMLnode* instance). The *insertBefore( )* method is similar to *appendChild( )* , but it lets us position a new node precisely in an existing XML object hierarchy.

Example

```
// Create a document with a P node
myDoc = new XML('<P>paragraph 2</P>');

// Create another P node and a text node
newP = myDoc.createElement("P");
newText = myDoc.createTextNode("paragraph 1");

// Append the text node to the new P node
newP.appendChild(newText);

// Insert the new P node (including its text child) before the existing P no
myDoc.insertBefore(newP, myDoc.firstChild);

trace(myDoc);  // Displays: "<P>paragraph 1</P><P>paragraph 2</P>"
```

See Also

XML.appendChild( ) `XML` `.previousSibling` , *XML.removeNode( )*

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Chapter 18. ActionScript Language Reference

XML.lastChild Property — Flash 5

a reference to the last descendant of a node — ***read-only***

theNode.lastChild

Description

The `lastChild` property is synonymous with `childNodes[childNodes.length - 1]`. It returns a reference to the last node object that descends from *`theNode`*, where *`theNode`* can be an *XML* or *XMLnode* instance. If *`theNode`* has no children, `lastChild` returns `null`.

In the following XML source fragment, the `lastChild` of the `MESSAGE` node is the `CONTENT` node:

```
<MESSAGE><USER>gray</USER><CONTENT>hi</CONTENT></MESSAGE>
```

Example

```
// Create a new XML document
myDoc = new XML('<MESSAGE><USER>gray</USER><CONTENT>hi</CONTENT></MESSAGE>')

// Sets msg to "hi" because myDoc's firstChild
// is MESSAGE; MESSAGE's lastChild is CONTENT; and CONTENT's firstChild
// is the text node with the value "hi"
msg = myDoc.firstChild.lastChild.firstChild.nodeValue
```

See Also

XML.childNodes, *XML*.firstChild, *XML*.nextSibling, *XML*.previousSibling

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Chapter 18. ActionScript Language Reference

XML.load( ) Method — Flash 5; security restrictions added in Flash 6

import XML source code from an external document

`xmlDoc.load(url)`

Arguments

url

A string specifying the location of the XML document to load.

Description

The *load( )* method imports an external XML document, parses it, converts it into an XML object hierarchy, and places that hierarchy into `xmlDoc`. Any previous contents of `xmlDoc` are replaced by the newly loaded XML content. The specified `xmlDoc` must be the top-level node in an XML object hierarchy (i.e., an instance of the *XML* class, not the *XMLnode* class).

Before accessing content imported with *load( )*, we must be sure the load and parsing operations are complete. To do so, either check the value of the XML document's `loaded` property, or assign the document an *onLoad( )* callback handler to respond to the load completion. See the `XML.loaded` and *XML.onLoad( )* entries for details. To determine whether the loaded data was parsed successfully, check the document's `status` property. To monitor the progress of the file transfer, use *getBytesLoaded( )* and *getBytesTotal( )*.

As of Flash 6, *XML.load( )* is subject to the domain-based security restrictions described under *System.security.allowDomain( )*.

Example

```
myDoc = new XML();
myDoc.load("myData.xml");
```

See Also

`XML.loaded`, *XML.onLoad( )*, *XML.sendAndLoad( )*, `XML.status`

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

XML.loaded Property

Flash 5

---

status of a load( ) or sendAndLoad( ) operation ***read-only***

`xmlDoc.loaded`

Description

The `loaded` property returns a Boolean value indicating whether a previously invoked *load( )* or *sendAndLoad( )* operation on `xmlDoc` has completed. The specified `xmlDoc` must be the top-level node in an XML object hierarchy (i.e., an instance of the *XML* class, not the *XMLnode* class).

The `loaded` property is immediately set to `false` when an XML *load( )* or *sendAndLoad( )* operation is initiated. If the load is successful, `loaded` is set to `true`; otherwise, it remains `false`. If no such operation has ever been executed on `xmlDoc`, `loaded` is `undefined`.

When `loaded` is `false`, the download and parsing of XML data is still in progress and attempts to access the object hierarchy in `xmlDoc` will fail. When `loaded` is `true`, XML data has finished being downloaded, parsed, and stored in `xmlDoc` as an object hierarchy. Note, however, that the loaded XML data may not have been parsed successfully (check the `status` property to determine if parsing was successful).

Example

The following example shows a basic XML preloader that waits for the XML data to be loaded before displaying it (XML preloaders can also be built using an *XML.onLoad( )* handler or *XML.getBytesLoaded( )* and *XML.getBytesTotal( )*):

```
// CODE ON FRAME 1
// Create a new XML document.
myDoc = new XML();
// Load an external XML file into the document.
myDoc.load("userProfile.xml");

// CODE ON FRAME 5
// If the data has loaded, go to the display frame.
// If not, loop back to frame 4 and then play.
// Loop until the data is done loading...
if (myDoc.loaded) {
  if (myDoc.status =  = 0) {
    gotoAndStop("displayData");
  } else {
    gotoAndStop("loadingError");
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

```
  }
} else {
  gotoAndPlay(4);
}
```

See Also

XML.load( ), XML.onLoad( ), XML.sendAndLoad( ), `XML.status`

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

XML.nextSibling Property

Flash 5

a reference to the node after this node ***read-only***

theNode.nextSibling

Description

The nextSibling property returns the node object after *theNode* in the current level of the XML object hierarchy (*theNode* can be an *XML* or *XMLnode* instance). If there is no node after *theNode*, nextSibling returns nu
In the following XML source fragment, the CONTENT node is the nextSibling of the USER node:

```
<MESSAGE><USER>gray</USER><CONTENT>hi</CONTENT></MESSAGE>
```

Example

Typically, the nextSibling property is used to *traverse* (move through or examine) an XML object hierarchy. For example, to view all the children of *theNode* in the order they appear, we can use:

```
for (var child = theNode.firstChild; child != null; child = child.nextSiblin
  trace("found node: " + child.nodeName);
}
```

By extending our loop into a function, we can recursively traverse every node in an XML object hierarchy, as follows:

```
function showNodes (node) {
  trace(node.nodeName + ": " + node.nodeValue);
  for (var child = node.firstChild; child != null; child = child.nextSibling
    showNodes(child);
  }
}

// Invoke the function on our node or document
showNodes(myDoc);
```

Note that in both traversal examples shown, text nodes show up without a name, as described under the nodeName entry

See Also

XML.childNodes, *XML*.firstChild, *XML*.lastChild, *XML*.nodeName, *XML*.nodeValue
XML.previousSibling

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

XML.nodeName Property

Flash 5

the name of the current node ***read/write***

`theNode.nodeName`

Description

The `nodeName` string property reflects the name of `theNode`, where `theNode` can be an *XML* or *XMLnode* instance. Since only two node types are supported by ActionScript (*element* nodes and *text* nodes), `nodeName` has only two possible values:

- If `theNode` is an element node, `nodeName` is a string matching the tag name of that element. For example, if `theNode` represents the element `<BOOK>`, then `theNode.nodeName` is `"BOOK"`.
- If `theNode` is a text node, `nodeName` is `null`. Note that this diverges from the DOM specification, which stipulates that `nodeName` for a text node should be the string `"#text"`. If you prefer, you can use the DOM-compliant `nodeType` property instead.

Example

We can use `nodeName` to check whether the current node has the element name we're seeking. For example, here we extract all the content of `H1` tags on the first level of an XML document (this example also checks for tags named `h1` with a lowercase `h`):

```
myDoc = new XML('<H1>first heading</H1><P>content</P>' +
                '<H1>second heading</H1><P>content</P>');
for (i = 0; i < myDoc.childNodes.length; i++) {
  if (myDoc.childNodes[i].nodeName.toUppercase() =  = "H1") {
    trace(myDoc.childNodes[i].firstChild.nodeValue);
  }
}
```

See Also

`XML.nodeType`, `XML.nodeValue`

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Chapter 18. ActionScript Language Reference

XML.nodeType Property — Flash 5

the type of the current node — ***read-only***

`theNode.nodeType`

Description

The `nodeType` is an integer property that returns `theNode`'s type (`theNode` can be an *XML* or *XMLnode* instance). Since only two node types are supported by ActionScript—*element nodes* and *text nodes*—`nodeName` has only two possible values: 1, if the node is an element node; and 3, if the node is a text node. These values may seem arbitrary, but actually they are the appropriate values as stipulated by the DOM. For reference, the other node types in the DOM are listed in Table 18-26.

Table 18-26. DOM node types

Node description	Node type code
ELEMENT_NODE[12]	1
ATTRIBUTE_NODE	2
TEXT_NODE[12]	3
CDATA_SECTION_NODE	4
ENTITY_REFERENCE_NODE	5
ENTITY_NODE	6
PROCESSING_INSTRUCTION_NODE	7
COMMENT_NODE	8
DOCUMENT_NODE	9
DOCUMENT_TYPE_NODE	10
DOCUMENT_FRAGMENT_NODE	11
NOTATION_NODE	12

[12] Supported by Flash.

Technically, ActionScript implements so-called *attribute*, *document*, and *document_type nodes* in addition to element and text nodes, but we don't have direct access to them as objects. For example, we can manipulate the attributes of a node through the `attributes` property, but we do not have direct access to attribute nodes themselves. Similarly, we have access to the `DOCTYPE` tag of a document through the `docTypeDecl` property, but we do not have direct access to the *document_type* node itself.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Element nodes correspond to XML or HTML tags. For example, in the XML fragment `<P>what is your favorite color?</P>`, the `P` tag is represented in an XML object hierarchy as an element node (`nodeType` 1). The text contained by a tag in XML source code—for example, the text "what is your favorite color?"—is represented as a text node (`nodeType` 3). CDATA section nodes are irreversibly converted to text nodes when they are parsed, as described under *XML.parseXML( )*.

Example

We can operate on a node conditionally, based on its `nodeType`. For example, here we remove all the empty text nodes that are children of *`theNode`*:

```
// Loop through all children of theNode
for (var i=0; i < theNode.childNodes.length; i++) {
  // If the current node is a text node...
  if (theNode.childNodes[i].nodeType =  = 3) {
    // Check for any useful characters in the node
    var emptyNode = true;
    for (var j=0; j < theNode.childNodes[i].nodeValue.length; j++) {
      // Useful character codes start above ASCII 32
      if (theNode.childNodes[i].nodeValue.charCodeAt(j) > 32) {
        emptyNode = false;
        break;
      }
    }
    if (emptyNode) {
      // No useful characters were found, so delete the node
      theNode.childNodes[i].removeNode();
    }
  }
}
```

See Also

The *XML* class, *`XMLnode`*`.attributes`, *`XML`*`.docTypeDecl`, *`XML`*`.nodeName`, *`XML`*`.nodeValue`

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

XML.nodeValue Property

Flash 5

the value of a node, as a string ***read/write***

theNode.nodeValue

Description

The nodeValue property reflects the string value of *theNode*, where *theNode* can be an *XML* or *XMLnode* instance. Since only two node types (element nodes and text nodes) are supported by ActionScript, nodeValue has only two possible values:

- If *theNode* is an element node, nodeValue is null.
- If *theNode* is a text node, nodeValue is the text contained by the node.

To assign new text to an existing text node, we use nodeValue as follows:

```
// Create a new XML document
myDoc = new XML('<H1>first heading</H1><P>content</P>');

// Change the text contained by the H1 tag
myDoc.firstChild.firstChild.nodeValue = "My Life Story";
```

To retrieve the text of a text node, we simply read its nodeValue property:

```
trace(myDoc.firstChild.firstChild.nodeValue);
```

Note that, in nodeValue, the predefined entities <, >, &, ", and ' are represented as the literal characters <, >, &, ", and '. To obtain the source code for a node, including uninterpreted entities, use *XML.toString( )*. For more details, see *XML.parseXML( )*.

When a node is used in a string context, Flash uses *XML.toString( )*, not nodeValue, to represent its value:

```
// Force an implicit datatype conversion. Calls toString( ).
trace(myDoc.firstChild.firstChild);
// Same as:
trace(myDoc.firstChild.firstChild.toString());
```

See Also

XML.nodeName, >, *XML.parseXML( )*, *XML.toString( )*

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Chapter 18. ActionScript Language Reference

XML.onData( ) Event Handler — Flash 5

executed when external XML source code finishes loading but has not yet been parsed

```
xmlDoc.onData(src);
```

Arguments

src

A string containing the loaded XML source code.

Description

The *onData( )* handler executes automatically whenever raw XML source has finished loading into `xmlDoc` due to an earlier *load( )* or *sendAndLoad( )* invocation. The specified `xmlDoc` must be the top-level node in an XML object hierarchy (i.e., an instance of the *XML* class, not the *XMLnode* class).

By default, *onData( )* has the following behavior:

- If the raw source received is `undefined`, it calls *xmlDoc.onLoad( )* with the `success` parameter set to `false`.
- Otherwise, it parses the source into `xmlDoc`, sets `xmlDoc.loaded` to `true`, and calls *xmlDoc.onLoad( )* with the `success` parameter set to `true`.

The *onData( )* handler can be assigned a custom callback function to intercept raw XML source code before ActionScript has a chance to parse it. Under certain circumstances, manipulating raw XML source manually may offer improved performance over ActionScript's built-in parsing, particularly in Flash Player 5. In Flash Player 6, the native XML parsing usually is faster than custom ActionScript routines.

Example

The following example shows how to display raw loaded XML source while preventing it from being parsed by ActionScript:

```
myDoc = new XML();
myDoc.onData = function (src) {
  trace("Here's the source: \n" + src);
};
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

```
myDoc.load("book.xml");
```

The following example intercepts and displays raw loaded XML source, but it also performs *onData( )*'s normal duties:

```
myDoc = new XML();
myDoc.onData = function (src) {
  trace("Here's the source: \n" + src);
  if (src =  = undefined) {
    this.onLoad(false);
  } else {
    this.parseXML(src);
    this.loaded = true;
    this.onLoad(true);
  }
};
myDoc.onLoad = function (success) {
  trace("Here's the parsed XML: " + this);
  trace("Load succeeded? " + success);
};
myDoc.load("book.xml");
```

See Also

XML.onLoad( ), XML.load( )

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Chapter 18. ActionScript Language Reference

XML.onLoad( ) Event Handler — Flash 5

executed when external XML data has been loaded and parsed

```
xmlDoc.onLoad(success)
```

Arguments

success

A Boolean value indicating whether loading was successful (`true`) or unsuccessful (`false`).

Description

The *onLoad( )* handler of `xmlDoc` executes automatically whenever an external XML file is loaded into `xmlDoc` via the *load( )* or *sendAndLoad( )* methods. The specified `xmlDoc` must be the top-level node in an XML object hierarchy (i.e., an instance of the *XML* class, not the *XMLnode* class).

By default, the *onLoad( )* handler of an XML document object is `undefined`. To use *onLoad( )*, we assign it a callback handler (i.e., a custom-made function). For example:

```
myDoc = new XML();
myDoc.onLoad = handleLoad;
function handleLoad (success) {
  // Process XML as desired here...
}
```

We rely on *onLoad( )* events to tell us when it's safe to process `xmlDoc`. If *onLoad( )* is triggered, we know that the loading and parsing of external XML data have completed, so we can safely access that loaded content. Hence, the *onLoad( )* handle alleviates the need to write preloading code to wait for data to arrive after the invocation of an XML *load( )* function. For example, in the following code we load an XML document and then wait for our custom *handleLoad( )* function to execute automatically when loading completes. If loading was successful, we process our XML content with the *displayProduct( )* function. Otherwise, we show an error message by executing the *displayError( )* function. (The *displayProduct( )* and *displayError( )* functions are custom functions that we've written to display information to the user, but they are not shown here.) Here is the code:

```
myDoc = new XML();
myDoc.onLoad = handleLoad;
myDoc.load("productInfo.xml");
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

```
function handleLoad(success) {
  if (success) {
    output = "Product information received";
    displayProduct(); // Call custom display function
  } else {
    output = "Attempt to load XML data failed";
    displayError(); // Call custom error display function
  }
}
```

Notice that we assign our *onLoad( )* callback before invoking *load( )*, which guarantees that the handler is available when the *load( )* operation completes.

The value of the `success` argument sent to *handleLoad( )* is set automatically by the interpreter to either `true` or `false`, indicating whether loading completed properly. In Flash 6, `success` is `false` for File Not Found (404) conditions and for security violations (see *System.security.allowDomain( )*). However, in Flash 5, the reception of an HTML error page from the server results in the parsing of that page into the target XML document object. Because the page parses properly, the load attempt is considered "successful" and `success` is set to `true`, even though the actual XML file may not have been found or some other server error may have been encountered.

In all cases, to be positive that you have the real data you requested, test its structure or content explicitly for some identifying characteristic, such as the `nodeName` of a particular child. Check `XML.status` to determine whether parsing succeeded. See also the *XML.onData( )* event handler, which can be used to perform custom parsing.

See Also

XML.load( ), *XML.onData( )*, *XML.sendAndLoad( )*, `XML.status`

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Chapter 18. ActionScript Language Reference

XML.parentNode Property — Flash 5

a reference to the immediate ancestor of a node — ***read-only***

`theNode.parentNode`

Description

The `parentNode` property returns a reference to the node object from which `theNode` descends in the XML object hierarchy, where `theNode` can be an *XML* or *XMLnode* instance. If `theNode` is the top (root) node of the current hierarchy, `parentNode` returns `null`.

In this XML source fragment, the `MESSAGE` node is the `parentNode` of text node "`hey`":

```
<MESSAGE>hey</MESSAGE>
```

Here, the `parentNode` of the `ROOM` node is the `HOTEL` node:

```
<HOTEL><ROOM><SIZE>Double</SIZE></ROOM></HOTEL>
```

See Also

`XML.childNodes`, `XML.firstChild`, `XML.lastChild`, `XML.previousSibling`

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

XML.parseXML( ) Method — Flash 5

parse a string of XML source code

```
xmlDoc.parseXML(string)
```

Arguments

string

A string of XML source code.

Description

The *parseXML( )* method reads and parses the XML source contained by `string`, converts that source into an object hierarchy, and then places the resulting hierarchy into `xmlDoc`. Any previous contents of `xmlDoc` are replaced by th new hierarchy. The specified `xmlDoc` must be the top-level node in an XML object hierarchy (i.e., an instance of the *XML* class, not the *XMLnode* class). Once parsing is complete, the success of the operation can be determined via `xmlDoc.status`.

To include raw HTML or XML source code in a text node without parsing it, use a CDATA section as follows:

```
<![CDATA[ source ]]>
```

For example, the following code creates a `MESSAGE` element with a single child text node containing the text "<B>Welcome</B> to my site" (the `<B>` tag is *not* interpreted as an XML tag and does not become part of the XML object hierarchy):

```
myDoc = new XML();
myDoc.parseXML("<MESSAGE><![CDATA[<B>Welcome</B> to my site]]></MESSAGE>");
```

Note that a CDATA section is interpreted as a node, so the following `<P>` element actually has three child nodes — a tex node, a CDATA node, and another text node:

```
<P>The following source: <![CDATA[<B>Welcome</B> to my site]]> adds bold.</P
```

However, in ActionScript, only text nodes and element nodes are supported, so this CDATA section is converted to a text node with a `nodeType` of 3 (text node), not 4 (CDATA section node). Once the original CDATA source code is parsed, it is not recoverable from the XML object hierarchy.

Regarding the use of the characters & and < in character data, the W3C XML 1.0 specification states:

> "The ampersand character (&) and the left angle bracket (<) may appear in their literal form only when used as

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

markup delimiters, or within a comment, a processing instruction, or a CDATA section. If they are needed elsewhere, they must be escaped using either numeric character references or the strings `&' and `<' respectively."[13]

[13] From "2.4 Character Data and Markup,"http://www.w3.org/TR/2000/REC-xml-20001006#syntax

For example, to include the code "`<P>We know that (10<11) is true.</P>`" in an XML document, we must use the `<` entity:

```
<P>We know that (10&lt;11) is true.</P>
```

or a CDATA section:

```
<P>We know that<![CDATA[ (10<11) ]]>is true.</P>
```

As of version 5.0.41.0 (5.0.42.0 on Internet Explorer), when Flash parses XML source containing a text node with the characters >, ", ', or &, it automatically converts them to the predefined entities `>` , `"` , `'` , and `&amp` . For example, in the following sample, the ' character becomes `'` :

```
// Apostrophe character is automatically converted to an entity
var doc = new XML("<p>don't worry</p>");
// Displays: "don&apos;t worry"
trace(doc.firstChild.firstChild.toString());
```

In contrast, a predefined entity in a text node is not changed during parsing. For example, when `&` is parsed it stays `&` — it is not converted to `&` or even `&` . However, when a predefined entity appears in a CDATA section, the initial ampersand is converted (that is, `&` *does* become `&)` . Therefore, literal characters, not entities, should always be used in CDATA.

The less-than character (<) is converted to `<` only when it appears in a CDATA section. Elsewhere, it must be writter as `<` to prevent a parsing error.

When the `nodeValue` of a text node is read, the predefined entities `<` , `>` , `"` , `'` , and `&` are automatically converted back to their corresponding characters:

```
var doc = new XML("<p>don&apos;t worry</p>");
// Displays: "don't worry"
trace(doc.firstChild.firstChild.nodeValue);
```

Note the difference: *XML.toString( )* of a text node returns the raw XML source of the node, with entities left in entity form; *XML* `.nodeValue` returns interpreted text suitable for output, with predefined entities converted to literal characters. For example:

```
doc = new XML("<P><![CDATA[what a 'nice' day]]></P>");
// Displays: "what a &apos;nice&apos; day"
trace(doc.firstChild.firstChild.toString());
// Displays: "what a 'nice' day"
trace(doc.firstChild.firstChild.nodeValue);
```

Usage

Note that in Flash 6, a carriage return in an attribute value causes a parsing error of status -8 (attribute not terminated). For loaded XML, offending returns can be replaced manually inside the *onData( )* handler.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Example

We can use *parseXML( )* as a means of replacing the current object hierarchy in an *XML* object with a new hierarchy based on internally composed XML source code (for example, some user input). In the following example, we create a simple XML message by combining markup with input from text fields named `username_txt` and `content_txt`

```
myDoc = new XML();
myXMLsource = "<MESSAGE><USER>" + username_txt + "</USER><CONTENT>"
    + content_txt + "</CONTENT></MESSAGE>";
myDoc.parseXML(myXMLsource);
```

See Also

XML `.ignoreWhite`, *XML.load ( )*, *XML* `.nodeValue`, *XML.onData( )*, *XML* `.status`, *XML.toString( )*

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Chapter 18. ActionScript Language Reference

XML.previousSibling Property — Flash 5

a reference to the node before this node — ***read-only***

theNode.previousSibling

Description

The previousSibling property returns a reference to the node object preceding *theNode* in the current level of the XML object hierarchy (*theNode* can be an *XML* or *XMLnode* instance). If there is no node before *theNode* in the current level of the hierarchy, it returns null.

In the following XML source fragment, the previousSibling of the CONTENT node is the USER node:

```
<MESSAGE><USER>gray</USER><CONTENT>hi</CONTENT></MESSAGE>
```

Example

The previousSibling property can be used to traverse an XML object hierarchy, although nextSibling is more commonly used for this purpose. To view all the children of *theNode* in reverse order, we can use:

```
for (var i = theNode.lastChild; i != null; i = i.previousSibling) {
  trace("found node: " + i.nodeName);
}
```

See Also

XML.childNodes, *XML*.firstChild, *XML*.lastChild, *XML*.nextSibling, *XML*.nodeName, *XML*.nodeValue, *XML*.parentNode

About Safari | Terms of Service | Privacy Policy | Contact Us | Help | Submit a Problem

Copyright © 2002 O'Reilly & Associates, Inc. All rights reserved.
1005 Gravenstein Highway North
Sebastopol, CA 95472

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Chapter 18. ActionScript Language Reference

XML.removeNode( ) Method — Flash 5

delete a node from an XML object hierarchy

theNode.removeNode( )

Description

The *removeNode( )* method deletes *theNode* from an XML document, where *theNode* can be an *XML* or *XMLnode* instance. All descendants (children, grandchildren, and so on) of *theNode* are also deleted. The childNodes property of *theNode*'s parent is automatically updated to reflect the new structure of the remaining object hierarchy.

Example

Here we delete the second child node; the third child node takes its place:

```
myDoc = new XML("<P>one</P><P>two</P><P>three</P>");
myDoc.childNodes[1].removeNode();
trace(myDoc);  // Displays: "<P>one</P><P>three</P>"
```

See Also

XML.appendChild( ), *XML.insertBefore( )*

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

XML.send( ) Method — Flash 5

send XML source code to an external script or application

```
xmlDoc.send(url, window)
```

Arguments

url

A string specifying the location of a script or application to which `xmlDoc` should be sent.

window

A required string specifying the name of the browser window or frame into which to load the script's response. It can be a custom name or one of the four presets: "`_blank`", "`_ parent`", "`_self`", or "`_top`". For details, see the description of the window settings under the *getURL( )* global function.

Description

The *send( )* method converts `xmlDoc` into a string of XML source code and sends that code in an HTTP request to the script or application residing at `url`. The specified `xmlDoc` must be the top-level node in an XML object hierarchy (i.e., an instance of the *XML* class, not the *XMLnode* class).

The receiving script or application at `url` is expected to process the XML in some way and, optionally, return a response—normally a web page—to the browser, which displays it in `window`. Note that the response is caught not by Flash but by the browser; to catch the response within Flash, use *sendAndLoad( )* instead.

When *send( )* is invoked from the Flash Player running in a browser, `xmlDoc` is sent via the POST method. When *send( )* is invoked from the Standalone Player on Windows, `xmlDoc` is sent via the GET method. (See Appendix F for details on support for GET and POST.) The server application receiving the posted XML string must be able to access the raw POST data of the HTTP request directly and should not attempt to parse it as normal name/value pairs. In Perl, the data in a POST request is available from STDIN, and can be extracted and stored in, say, `$buffer` as follows:

```
read(STDIN,$buffer,$ENV{'CONTENT_LENGTH'});
```

In ASP, raw POST data can be accessed via the *Request.BinaryRead* method.

In PHP 4.1 and later, the following option in *php.ini* forces raw POST data to be populated regardless of the MIME type of the request:

```
always_populate_raw_post_data = On
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

With this option enabled, `$HTTP_RAW_POST_DATA` will be set for every POST request. Some applications (e.g., versions of Cold Fusion prior to ColdFusion MX) may not have a direct means of accessing the data in a POST request. For these situations, it may be necessary to first convert the XML object to a string using *XML.toString( )* and then pass that string to the server as a variable using *loadVariables( )*.

The default MIME content type of the XML text sent to the server is "`application/x-www-form-urlencoded`". This type, however, is only cosmetic—the text itself is not URL-encoded. In Flash Player 5.0.41.0 and later, the MIME content type can be modified using the `contentType` property. For example, to set the MIME type to "`application/xml`", we use:

```
xmlDoc = new XML();
xmlDoc.contentType = "application/xml";
```

Nevertheless, setting the `contentType` property explicitly to "`application/x-www-urlform-encoded`" still does not cause the text sent to be URL-encoded.

Note that in Flash Player 5.0.41.0, when XML source is parsed and the characters `&`, ', ", and `<` appear in a text node, they are converted to the following entities: `&`, `'`, `"`, and `>`. Within Flash, the entities are converted back to their original characters when the `nodeValue` of an element is accessed; however, the entities will show up in XML source sent to the server.

Example

```
theDoc = new XML("<SEARCH_TERM>tutorials</SEARCH_TERM>");
theDoc.send("http://www.domain.com/cgi-bin/lookup.cgi", "remoteWin");
```

See Also

XML.load( ), *`XML`*`.loaded`, *XML.onLoad( )*, *XML.sendAndLoad( )*, *`XML`*`.status`

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

XML.sendAndLoad( ) Method — Flash 5

send XML source code to an external script or application and receive XML source in return

```
xmlDoc.sendAndLoad(url, resultXML)
```

Arguments

url

A string specifying the location of a script or application to which `xmlDoc` should be sent.

resultXML

A reference to an XML document object that will receive the returned XML source code.

Description

The *sendAndLoad( )* method serializes `xmlDoc` into a string of XML source code and sends that string to a script or application that resides at `url` . The specified `xmlDoc` must be the top-level node in an XML object hierarchy (i.e., an instance of the *XML* class, not the *XMLnode* class).

The receiving script or application at `url` is expected to process the XML in some way and send an XML document back response. The response document is caught by Flash, parsed, converted into an XML object hierarchy, and placed in `resultXML` . Any previous contents of `resultXML` are replaced by the newly loaded XML content. See *XML.send* for important information about sending XML to a server. Note that unlike *send( )* , *sendAndLoad( )* sends `xmlDoc` usi POST request in both the browser and Standalone versions of Flash Player 6. However, in the Standalone Flash Player 5 c Windows, *sendAndLoad( )* uses the GET method. See Appendix F for details on support for GET and POST.

Usage

Before accessing content imported with *sendAndLoad( )* , we must be sure that the loading and parsing operations are complete. To do so, we either check the value of `resultXML` 's `loaded` property or we assign `resultXML` an *onl)* event handler to respond to the load completion. See the `XML` `.loaded` and *XML.onLoad( )* entries for details. To determine the result of parsing the loaded data, we check the document's `status` property.

XML.sendAndLoad( ) is subject to the domain-based security restrictions described under *System.security.allowDomain(*

Example

```
// Create an XML document.
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

```
myDoc = new XML("<P>hello server!</P>");

// Create an empty XML document to receive the server's response.
serverResponse = new XML();

// Send myDoc to the server, and place the response in serverResponse.
myDoc.sendAndLoad("http://www.domain.com/cgi-bin/readData.cgi", serverRespon

// Add an onLoad( ) handler to serverResponse that displays the response
// from the server in the text field output.
serverResponse.onLoad = function () {
  output = serverResponse.toString();
}
```

See Also

XML.load ( ) , XML .loaded , *XML.onData( )* , *XML.onLoad( )* , *XML.send( )* , XML .status

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Chapter 18. ActionScript Language Reference

XML.status Property — Flash 5

indicates whether parsing XML source into an object hierarchy was successful — ***read-only***

xmlDoc.status

Description

The `status` property returns a numeric status code indicating whether any errors were encountered when parsing XML source code. The specified *xmlDoc* must be the top-level node in an XML object hierarchy (i.e., an instance of the *XML* class, not the *XMLnode* class).

Parsing occurs when source XML is:

- Provided as an argument to the *XML( )* constructor
- Passed to the *parseXML( )* method
- Loaded into a new *XML* object via the *load( )* or *sendAndLoad( )* methods

The `status` codes are shown in Table 18-27. If no errors were encountered in parsing, success is indicated by a `status` of 0. Errors are indicated by negative numbers. Parsing terminates once the first error is encountered, so other errors may surface even after you address previously reported errors.

Table 18-27. XML parsing status codes

Status	Description
0	The document parsed without errors (i.e., success).
-2	A CDATA section was not properly terminated.
-3	The XML declaration was not properly terminated.
-4	The `DOCTYPE` declaration was not properly terminated.
-5	A comment was not properly terminated.
-6	An XML element was malformed.
-7	Not enough memory to parse the XML source.
-8	An attribute value was not properly terminated.
-9	A start tag had no corresponding end tag.
-10	An end tag had no corresponding start tag.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Normally, we use `status` to determine whether it's safe to proceed with processing an externally loaded XML file. Check the `loaded` property to ensure that a *load( )* or *sendAndLoad( )* command has completed before checking the `status`. Note that ActionScript's XML parser does not validate documents against DTDs; it only verifies well-formedness.

Example

```
myDoc = new XML("<BOOK>Colin Moock</AUTHOR></BOOK>");
trace(myDoc.status);  // Displays: "-10" (missing start tag)
```

See Also

XML.load( ), *XML*`.loaded`, *XML.onLoad( )*, *XML.parseXML( )*, *XML.sendAndLoad( )*

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Chapter 18. ActionScript Language Reference

XML.toString( ) Method Flash 5

the source code of the XML node, as a string

```
theNode.toString()
```

Returns

A string representing the source code of the XML object hierarchy starting at `theNode` .

Description

The *toString( )* method converts an XML node object or an XML document object to its analogous XML source code (`theNode` can be an *XML* or *XMLnode* instance). If `theNode` is a top-level XML document object, any `DOCTYPE` XML declaration tags are included in the string. If the document's `ignoreWhite` property is `false` , whitespace is preserved and the document source code appears as it did when it was parsed.

Note that the predefined entities `<` , `>` , `&` , `"` , and `'` are not converted to literal characters *toString( )* . To obtain the text value of a node with entities converted to literal characters, use `nodeValue` . For details, *parseXML( )* .

Normally, it's not necessary to invoke *toString( )* explicitly; *toString( )* is automatically invoked any time `theNode` is u in a string context.

Example

```
var myDoc = new XML('<?xml version="1.0"?><!DOCTYPE foo SYSTEM "bar.dtd"><BO
<TITLE>ActionScript for Flash MX: The Definitive Guide</TITLE>'
  + '<AUTHOR SALUTATION="Mr.">Colin Moock  </AUTHOR>    '
  + '<PUBLISHER>O\'Reilly & Associates, Inc</PUBLISHER>    </BOOK>');

trace(myDoc.toString());
// Displays:
<?xml version="1.0"?><!DOCTYPE foo SYSTEM "bar.dtd">
<BOOK>       <TITLE>ActionScript for Flash MX:
 The Definitive Guide</TITLE><AUTHOR SALUTATION="Mr.">Colin
Moock  </AUTHOR>    <PUBLISHER>O'Reilly & Associates, Inc
</PUBLISHER>    </BOOK>
```

See Also

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Object.toString( ) , *XML* .nodeValue

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Chapter 18. ActionScript Language Reference

XML.xmlDecl Property — Flash 5

the document's XML declaration tag — ***read/write***

xmlDoc.xmlDecl

Description

The string `xmlDecl` property represents the XML declaration tag of *`xmlDoc`*, if any exists. Otherwise, `xmlDecl` is `undefined`. The specified *`xmlDoc`* must be the top-level node in an XML object hierarchy (i.e., an instance of the *XML* class, not the *XMLnode* class).

The XML declaration tag of an XML document is used to identify the version of XML being used in the document. We use the XML declaration tag to build well-formed XML documents that can be validated externally.

Example

```
// A well-formed document (but not validated against a DTD)
myXML = new XML('<?xml version="1.0"?><P>this is a short document</P>');
trace(myXML.xmlDecl);  // Displays: "<?xml version="1.0"?>"
// Set a new XML declaration
myXML.xmlDecl = '<?xml version="1.0" standalone="no"?>';
```

See Also

XML.docTypeDecl

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Chapter 18. ActionScript Language Reference

XMLnode Class

internal superclass of the XML classFlash 5 (undocumented)

Description

The *XMLnode* class defines the core properties and methods of nodes in an XML object hierarchy. Though *XMLnode* is internal to ActionScript, it can be used by programmers to add to the default functionality of *XML* objects.

Technically, every XML object hierarchy includes two kinds of object nodes:

- One *XML* node, which serves as the main container for the hierarchy
- An arbitrary number of *XMLnode* nodes, which are the children of the main container node

The main container node is an instance of the *XML* class. For example, if we create `myDoc` as follows:

```
myDoc = new XML();
```

then `myDoc` is an instance of the *XML* class. The *XML* class inherits from the *XMLnode* class, so main container nodes all the properties and methods defined by *XMLnode* plus those defined by *XML* . In contrast, the children of `myDoc` will actually be instances of the *XMLnode* class, not the *XML* class.

So, if we create `myParagraph` as follows:

```
myParagraph = myDoc.createElement("P");
```

then `myParagraph` is an instance of the *XMLnode* class. Most of the time, the internal distinction between node classe does not affect our use of XML objects. However, if we wish to add an inherited property or method to all XML objects, t we must use the *XMLnode* class's `prototype` , not the *XML* class's `prototype` (see the following Example). Any methods or properties attached to *`XMLnode`* `.prototype` are inherited by all XML nodes in a movie.

For reference, the properties, methods, and event handlers defined by *XMLnode* and *XML* are listed in Table 18-28. Note while all listed items are accessible through instances of the *XML* class, items defined by *XML* are not available through instances of *XMLnode* . For example, the *load( )* method can be invoked on an instance of the *XML* class, but not on an instance of the *XMLnode* class. For a full discussion of each item, see the appropriate *XML* class entry.

Table 18-28. XMLnode and XML properties, methods, and event handlers

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

XMLnode and XML	XML only
appendChild( )	`contentType`
`attributes`	*createElement( )*
`childNodes`	*createTextNode( )*
cloneNode( )	`docTypeDecl`
`firstChild`	*getBytesTotal( )*
hasChildNodes( )	*getBytesLoaded( )*
insertBefore( )	`ignoreWhite`
`lastChild`	*load( )*
`nextSibling`	`loaded`
`nodeName`	*onData( )*
`nodeType`	*onLoad( )*
`nodeValue`	*parseXML( )*
`parentNode`	*send( )*
`previousSibling`	*sendAndLoad( )*
removeNode( )	`status`
toString( )	`xmlDecl`

Example

The following code adds a custom *secondChild( )* method to *`XMLnode`* `.prototype` (the *secondChild( )* method is subsequently available from any XML node in our movie):

```
XMLnode.prototype.secondChild = function () {
  return this.childNodes[1];
};

myDoc = new XML("<PRODUCT>Cell Phone</PRODUCT><PRODUCT>Game Console</PRODUCT

trace(myDoc.secondChild());  // Displays: "<PRODUCT>Game Console</PRODUCT>"
```

It's also perfectly legitimate to augment the *XML* class via *`XML`* `.prototype` , but such additions apply only to main container nodes (direct instances of the *XML* class).

See Also

The *XML* class; "Superclasses and Subclasses," in Chapter 12

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Chapter 18. ActionScript Language Reference

XMLSocket Class — Flash 5

support for a persistent client/server TCP/IP connection

Constructor

`new XMLSocket()`

Methods

close ( )

Terminate an open connection to a server application.

connect ( )

Attempt to establish a new connection to a server application.

send ( )

Send an XML object hierarchy to a server application as a string.

Event handlers

onClose ( )

Executes when the server terminates the connection.

onConnect ( )

Executes when a connection attempt completes.

onData ( )

Executes when data is received but has not yet been parsed as XML.

onXML ( )

Executes when data has been received and parsed into an XML object hierarchy.

Description

The majority of connections between Flash and a server have a very short life span. When Flash requests external data via *loadMovie( )* , *LoadVars* , or *XML* , a temporary communication channel is established. Data is sent over that channel, an then the channel is terminated. This kind of short-term communication has many useful applications, but it is also limited two important ways:

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

- Once the connection closes, the server has no way of contacting Flash. Flash must always initiate communication w the server.

- Each time Flash obtains information from the server, a new connection must be opened. The time and processor loa involved in opening connections repeatedly prevents Flash from engaging in anything near real-time transactions w server.

As of Flash 5, we can overcome these limitations with the *XMLSocket* class, which allows us to open a persistent communication link between a server application and Flash. We use *XMLSocket* to develop systems that require frequent server updates, such as a chat room or a networked multiplayer game.

In order to connect to a remote application using *XMLSocket* , we must first create and store an *XMLSocket* object, like t

```
mySocket = new XMLSocket();
```

Next, we invoke the *connect( )* method, which asks Flash to establish a communication link with the remote application. F example:

```
mySocket.connect("http://www.myserver.com", 8000);
```

Once a connection is established, the *XMLSocket* object acts as a transmitter/receiver. We send XML-formatted data to th remote application by invoking the socket's *send( )* method, and we know that we've received XML-formatted data when t socket's *onXML( )* event is triggered.

A server application used with an *XMLSocket* object must:

- Serve TCP/IP socket connections on a specific port greater than or equal to 1024

- Transmit XML-formatted data in segments delimited by a *zero byte* (i.e., the ASCII null character)

Typically, server applications are created by server-side programmers, not Flash programmers. Some of the available Flas compatible socket servers are listed in Table 18-29.

Table 18-29. Flash-compatible socket servers

Name	Description	License	Location
moockComm	A light-weight reference implementation, written in Java	Free	http://www.moock.org/chat/moockComm.zip
aquaServer	A light-weight reference implementation, written in Java	Free	http://www.figleaf.com/development/flash5/aquaserver
Unity	A full-featured rooms-based server written in Java	Pay	http://www.moock.org/unity
Fortress	A mature, heavy-duty socket server	Pay	http://www.xadra.com/products/main.html
ElectroServer	A full-featured, real-world tested socket server	Pay	http://www.electrotank.com/ElectroServer

An *XMLSocket* connection stays open until one of the following occurs:

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

- The *close ( )* method of the *XMLSocket* object is called.
- No more references to the *XMLSocket* object exist.
- The server terminates the connection (this triggers an *onClose ( )* event).
- The movie is unloaded or the Flash Player exits.

The *XMLSocket* class also provides four event handlers—*onClose( )* , *onConnect( )* , *onData( )* , and *onXML( )* —that all us to monitor the status of an open connection.

If you fail to define callback handlers for the `onClose` and `onConnect` properties, you won't be able to perform any error checking or provide any feedback about a connection attempt. If you fail to define a callback handler for your `onXML` property, you won't be notified when the socket receives data from a server-side application, nor will you be able to retrieve such data.

For extensive information on *XMLSocket* development, see:

http://www.moock.org/unity
http://hotwired.lycos.com/webmonkey/multimedia/shockwave_flash/tutorials/tutorial9.html

Note that as of Flash 6, real-time communication can also be implemented with Macromedia Flash Communication Serve MX (Comm Server). Comm Server does not use XML; rather, it uses the custom RMTP protocol for seamless remote execution of ActionScript code. Furthermore, Comm Server supports video and audio streaming. For more information, se

http://www.macromedia.com/software/flashcom/

Example

Example 18-10 defines an *XMLSocket* subclass, *SocketController* , which wraps the use of the *XMLSocket* class into a s of convenient methods and event handlers. The class is saved in the file *SocketController.as* and is available at the online Code Depot. Example 18-11 shows how to use *SocketController* to create a very simple chat application.

Example 18-10. The SocketController class

```
/*
 * SocketController Class. Extends XMLSocket.
 *   Version: 1.0.0
 *   Desc: Provides services for communicating with a socket server.
 *
 * Methods
 *  doConnect()              -connect to server
 *  setServer()              -set server host/port
 *  killConnect()            -disconnect from server
 *  send()                   -send XML to the server
 *
 * Event Handlers
 *  onConnectFailure()       -when connect attempt fails
 *  onConnectSuccess()       -when connect attempt succeeds
 *  onServerKillConnect()    -when server kills connection
 *  onClientKillConnect()    -when Flash kills connection
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

```
 *  onXML()                    -when new XML arrives
 */

// Set superclass to XMLSocket.
SocketController.prototype = new XMLSocket();

// Class constructor
function SocketController (host, port) {
  // Instance properties
  this.host = null;
  this.port = 0;

  // Initialization: Assign provided values to host and port properties.
  if (arguments.length > 1) {
    this.setServer(host, port);
  }
}

// Instance Methods
/*
 * Attempt to connect to the server.
 */
SocketController.prototype.doConnect = function () {
  // If there's no valid server defined, then fail
  if (this.host =  = null || this.port < 1024) {
    trace("Connection attempt aborted: invalid host or port.");
    this.onConnectFailure();
    return false;
  } else {
    // Otherwise, attempt to connect (connect returns true if the
    // initial stage of connection succeeds).
    var connectSuccess = this.connect(this.host, this.port);
  }

  // If the initial connection stage succeeds...
  if (connectSuccess) {
    // ...wait for the onConnect() event to fire.
    trace("Initial connection succeeded. Awaiting server response...");
  } else {
    // ...otherwise, connect() returned false, so we didn't
    // establish a connection.
    trace("Initial connection failed");
    // Invoke the user-defined callback for connect failure.
    this.onConnectFailure();
  }
};

/*
 * Set the host and port of the server.
 */
SocketController.prototype.setServer = function (host, port) {
  // Error handling. Host must be a string,
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

```
    // and port must be a number over 1023.
    if (typeof host != "string"
        || typeof port != "number"
        || (typeof port =  = "number" && port < 1024)) {
      trace("Server assignment failed: invalid host or port.");
      // Reset host and port.
      this.host = null;
      this.port = 0;
      return false;
    }

    // Host and port are valid, so assign them.
    trace("Server set to: " + host + " " + port);
    this.host = host;
    this.port = port;
};

/*
 * Callback to respond to the completion of a connection attempt.
 */
SocketController.prototype.onConnect = function (success) {
  // If onConnect( )'s success argument is true,
  // connection has been established.
  if (success) {
    trace("Connection to: " + this.host + " " + this.port + " established");
    // Invoke the user-defined callback for connection success.
    this.onConnectSuccess();
  } else {
    // success was false, so the connection was not established.
    trace("Connection to: " + this.host + " "
                        + this.port + " could not be established");
    // Invoke the user-defined callback for connection failure.
    this.onConnectFailure();
  }
};

/*
 * Callback invoked when server kills the connection.
 */
SocketController.prototype.onClose = function () {
  trace("The server has terminated the connection.");
  // Invoke the user-defined callback for server disconnect
  this.onServerKillConnect();
};

/*
 * Closes the connection to the server manually.
 */
SocketController.prototype.killConnect = function () {
  trace("Closing current connection.");
  this.close();
  // Invoke the user-defined callback for client disconnect
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

```
  this.onClientKillConnect();
};
```

Example 18-11 demonstrates the use of *SocketController* with a simple chat application.

Example 18-11. A simple SocketController chat app

```
// A Simple Chat App
// Load SocketController class
#include "SocketController.as"
// Start the show
main();

// The chat application
function main () {
  // Store a portable reference to the current timeline
  _global.chatMain = this;

// Create sending text field
chatMain.createTextField("send_txt", 1, 100, 100, 300, 20);
chatMain.send_txt.border = true;
chatMain.send_txt.type = "input";

// Create output text field
chatMain.createTextField("output_txt", 2, 100, 130, 300, 200);
chatMain.output_txt.border = true;
chatMain.output_txt.type = "dynamic";

// Create the SocketController instance and connect to the server
chatController = new SocketController();
chatController.setServer("moock.org", 8050);
chatController.doConnect();

// Invoked when a message is received
chatController.onXML = function (src) {
  trace("Received XML: " + src);
  var message = src.firstChild.firstChild.nodeValue;
  chatMain.output_txt.text +=  message + "\n";
};

// Invoked when a connection attempt fails
chatController.onConnectFailure = function () {
  chatMain.output_txt.text +=  "Connection attempt failed." + "\n";
};

// Invoked when a connection attempt succeeds
chatController.onConnectSuccess = function () {
  chatMain.output_txt.text +=  "Connection established." + "\n";
};

// Invoked when the server drops the connection
chatController.onServerKillConnect = function () {
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

```
  chatMain.output_txt.text +=  "Connection terminated by server." + "\n";
};

// Invoked when Flash closes the connection
chatController.onClientKillConnect = function () {
  chatMain.output_txt.text +=  "Connection closed." + "\n";
};

// Add a listener to capture Enter key presses
chatController.onKeyDown = function () {
  // If the Enter key is pressed while the cursor is in the send field...
  if (Key.getCode() =  = Key.ENTER
      && eval(Selection.getFocus()) =  = chatMain.send_txt) {
    // ...send the contents of the send field in a <MESSAGE> tag
    chatController.send("<MESSAGE>" + chatMain.send_txt.text + "</MESSAGE>

    // Then clear the input field
    chatMain.send_txt.text = "";
  }
};
// Register the listener
Key.addListener(chatController);
}
```

See Also

loadVariables( ) , the *LoadVars* class, the *XML* class

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Chapter 18. ActionScript Language Reference

XMLSocket.close( ) Method Flash 5

terminate an open connection to a server application

```
socket.close()
```

Description

The *close( )* method severs the communication link between `socket` and a server application. Once *close( )* has been executed on `socket`, subsequent attempts to invoke *send( )* on `socket` fail. Likewise, the server application will no longer be able to send data to Flash through `socket`.

Note that *close( )* has no effect if the `socket` is already closed or was never connected. Furthermore, *close( )* does not trigger the *onClose( )* handler of the socket object—*onClose( )* is triggered only by a server-side connection closure.

See Also

XMLSocket.connect( ), XMLSocket.onClose( ), XMLSocket.send( )

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Chapter 18. ActionScript Language Reference

XMLSocket.connect( ) Method — Flash 5

open a connection to a server application

```
socket.connect(host, port)
```

Arguments

host

A string specifying a hostname, such as "`www.myserver.com`", or a standard IP address (four dot-separated, 8-bit, decimal integers, such as 111.222.3.123). If `null` or an empty string is specified, it defaults to the server address from which the movie was served.

port

An integer specifying a TCP port number greater than or equal to 1024. There is no standard port number reserved (nor is there a widespread convention) for *XMLSocket* communications.

Returns

A Boolean indicating the initial success (`true`) or failure (`false`) of the connection attempt.

Description

The *connect( )* method attempts to establish a connection from Flash to a server application running on `host` at `port`.

If *connect( )* returns `true`, the initial phase of the connection completed successfully and the `socket`'s *onConnect( )* handler will be invoked at a later time. From the *onConnect( )* handler, we can evaluate whether the connection was fully established. Note that connection attempts can take varying amounts of time, particularly when a connection attempt fails. You should indicate to the user that a connection attempt is in progress when invoking *connect( )*.

If *connect( )* returns `false`, the initial connection phase did not complete successfully. In such a case, `socket`'s *onConnect( )* handler will not be invoked.

It is important to check both the return value of the *connect( )* method and, if *connect( )* returns `true`, the value of the `success` parameter sent to the *onConnect( )* handler.

Usage

For security reasons, *connect( )* is not permitted to connect to arbitrary Internet hosts. It can connect only to hosts in the domain from which the movie was downloaded. The cross-domain access rules for *connect( )* are described under *System.security.allowDomain( )*. The *connect( )* method returns `false` for connection attempts that violate security

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

restrictions. Note that security restrictions do not apply to the Standalone Player.

Example

```
// Create a new socket object
mySocket = new XMLSocket();
// Assign a callback handler function to onConnect
mySocket.onConnect = handleConnect;
// Attempt to connect to an application running on myserver.com at port 1000
if (mySocket.connect("myserver.com", 10000) =  = false) {
  // Jump to a frame showing some sort of error message
  this.gotoAndStop("failureDialog");
} else {
  // Jump to a frame where we'll wait until onConnect( ) is triggered
  this.gotoAndPlay("connecting");
}
```

See Also

XMLSocket.close( ) , XMLSocket.onConnect( )

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Chapter 18. ActionScript Language Reference

XMLSocket.onClose( ) Event Handler Flash 5

invoked when the server closes the connection

socket.onClose()

Description

The *onClose( )* event handler is executed whenever an open connection to *socket* is closed by the server (not by Flash). Server-instigated closures usually result from a server application shutting down or deliberately "kicking off" the client.

Example

To respond to an *onClose( )* event, we must assign our own function (i.e., our callback handler) to the onClose property of an *XMLSocket* object. We use the callback handler to detect an external socket disconnection. The following code assigns the function *handleClose( )* to mySocket's onClose property. The *handleClose( )* function simply alerts the user that a closure has occurred by updating the value of the text field status_txt:

```
mySocket = new XMLSocket();
mySocket.onClose = handleClose;

function handleClose () {
  status_txt.text += ("\nThe server has terminated the connection.\n");
}
```

See Also

XMLSocket.close( ); Chapter 10

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Chapter 18. ActionScript Language Reference

XMLSocket.onConnect( ) Event Handler Flash 5

invoked when a connection attempt is completed, successfully or otherwise

```
socket.onConnect(success)
```

Arguments

success

A Boolean indicating whether the connection attempt succeeded (`true`) or failed (`false`).

Description

The *onConnect( )* event handler is executed when a previously invoked *connect( )* operation finishes. The execution of *onConnect( )* does not necessarily mean a connection has been successfully established—it is executed whenever the connection *attempt* finishes, whether or not the attempt was successful. The callback handler specified by *onConnect( )* is passed a `success` argument that indicates if the attempt succeeded (i.e., a connection has been established). If so, `success` is set to `true`. If the attempt failed (i.e., a connection timed out, was refused, or otherwise could not be established), the `success` argument is set to `false`. Note that ActionScript does not distinguish among network timeout, unknown host, refusal, or other common connection errors. Therefore, the callback handler may not be executed for up to a minute after the *connect( )* command is issued, depending on the settings of the server involved in a connection attempt, the connection speed, network traffic, and so on.

Example

We use *onConnect( )* to detect the success or failure of a connection attempt. In practice, we might use it to set a flag indicating that transmissions should begin if the connection was successful. We may also use *onConnect( )* to execute fallback code when the connection fails, such as alerting the user to the problem's nature.

To respond to an *onConnect( )* event, we must assign a custom function (i.e., our callback handler) to the `onConnect` property of an *XMLSocket* object. The following code assigns the function *handleConnect( )* to `mySocket`'s `onConnect` property. By updating the value of the text field `status_txt`, *handleConnect( )* alerts the user that a connection has either succeeded or failed:

```
mySocket = new XMLSocket();
mySocket.onConnect = handleConnect;

function handleConnect (succeeded) {
  if (succeeded) {
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

```
    status_txt.text += ("Successfully connected.\n");
  } else {
    status_txt.text += ("Connection attempt failed.\n");
  }
}
```

For code showing the *onConnect( )* handler used in a more complete system, see Example 18-11 under the *XMLSocket* class.

See Also

XMLSocket.connect( ), *XMLSocket.onClose( )*; Chapter 10

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Chapter 18. ActionScript Language Reference

XMLSocket.onData( ) Event Handler — Flash 5

executed when external data is received but has not yet been parsed as XML

```
socket.onData(src)
```

Arguments

src

A string containing loaded data, which is usually XML source code.

Description

The *onData( )* event handler executes whenever a zero byte (ASCII null character) is transmitted to Flash over `socket`. By default, *onData( )* simply constructs a new XML object hierarchy from `src` and passes that hierarchy to *socket.onXML( )*. However, the *onData( )* handler can be assigned a custom callback function to intercept `src` before ActionScript has a chance to parse it as XML. Under certain circumstances (such as realtime video games), manipulating the raw data in `src` manually may offer improved performance over ActionScript's built-in XML parsing.

Example

The following code shows how to assign a custom callback function to *onData( )*. The callback function simply displays any data received by `mySocket` and prevents ActionScript from parsing the data as XML:

```
mySocket = new XMLSocket();

mySocket.onData = function (src) {
  trace("Received data: \n" + src);
};
```

See Also

XMLSocket.onXML( )

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Chapter 18. ActionScript Language Reference

XMLSocket.onXML( ) Event Handler — Flash 5

invoked when data is received by an XMLSocket object and has been parsed as XML

socket.onXML(*xmlObject*)

Arguments

xmlObject

The XML object that will house the incoming XML-formatted data.

Description

The *onXML( )* event handler executes when Flash receives, and has parsed, an incoming transmission. Whenever *socket* receives a complete block of data (i.e., a string followed by a `null` character) from the server, the callback function specified by *socket*.onXML is automatically invoked. A server can send data as often as it pleases, but the callback function is executed only when the trailing `null` character (i.e., a zero byte) is received by *socket*. In Java, a zero byte is specified as `'\0'`.

When the zero byte is received, it causes ActionScript to parse any data that has been received by *socket* since the last zero byte was sent (or since the initial connection, if this is the first zero byte). The parsed data is converted to an XML object hierarchy, which is passed as the *xmlObject* argument to the callback function.

If you are a Flash programmer who is responsible for only the client side of a client/server application, simply note that the callback function specified by `onXML` receives any new XML data when it arrives. The new XML data is accessible through *xmlObject*.

To access the raw data sent over a socket, override the default behavior of the socket's *onData( )* handler (see *XMLSocket.onData( )*).

Example

To respond to an *onXML( )* event, we must assign our own callback function to the `onXML` property of an *XMLSocket* object. The following code assigns the function *handleIncoming( )* to `mySocket`'s `onXML` property. The *handleIncoming( )* function accesses one of the nodes of the XML object hierarchy stored in `messageObj` and adds its value to the text field `messages_txt`:

```
mySocket = new XMLSocket();
mySocket.onXML = handleIncoming;
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

```
function handleIncoming (messageObj) {
  trace("Got some new data!");
  // messageObj will contain the fragment: <MESSAGE>text</MESSAGE>
  var message = messageObj.firstChild.firstChild;
  messages_txt.text += (message.nodeValue + "\n");
}
```

For code showing the *onXML( )* handler used in a more complete system, see Example 18-11 under the *XMLSocket* class.

See Also

XMLSocket.send( ), *XMLSocket.onData( )*; Chapter 10

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Chapter 18. ActionScript Language Reference

XMLSocket.send( ) Method — Flash 5

transmit XML-formatted data to a server application

```
socket.send(xmlObject)
```

Arguments

xmlObject

An *XML* object to be converted into a string and sent to the server application or any string containing XML-formatted text.

Description

The *send( )* method transmits a message from Flash to a server application via `socket`. The message to send should be an instance of the *XML* class but may also be an XML-formatted string. When *send( )* is invoked, `xmlObject` is converted to a string and sent to the remote application, followed by a zero byte (ASCII null). The remote application is not obliged to respond; however, any response sent will trigger `socket`'s *onXML( )* event handler.

Example

The following code sends a very simple XML-formatted message to a remote application over the socket `mySocket`, which is a valid *XMLSocket* object for which a connection has already been established (note that `message` is an *XML* object, not an *XMLSocket* object; see Example 18-11 under the *XMLSocket* class entry for a full-fledged *XMLSocket* sample application):

```
var message = new XML('<MESSAGE>testing...testing...</MESSAGE>');
mySocket.send(message);
```

It is also legal to send a string containing XML-formatted text without wrapping it in an *XML* object. For simple XML messages, this is often sufficient:

```
mySocket.send('<MESSAGE>testing...testing...</MESSAGE>');
```

See Also

XMLSocket.onXML( ), *XML.send( )*; the *XMLSocket* class

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Part III: Appendixes

- Appendix A
- Appendix B
- Appendix C
- Appendix D
- Appendix E
- Appendix F
- Appendix G
- Appendix H

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Part III: Appendixes

Appendix A. Resources

The following are resources of interest. See also the URLs cited in Table P-3 in the Preface.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Appendix A. Resources

A.1 ActionScript Books

Among the hundreds of Flash and ActionScript books on the market, the following books are, for me, the few that stand above the crowd:

Object-Oriented Programming with ActionScript, Branden Hall and Samuel Wan. New Riders Publishing, 2002.
Flash MX Application Design and Development, Jessica Speigel. New Riders Publishing, 2002.
Robert Penner's Programming Macromedia Flash MX, Robert Penner. Osborne McGraw-Hill, 2002.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Appendix A. Resources

A.2 ActionScript and Programming Web Sites

ActionScript for Flash MX: The Definitive Guide book site

http://www.moock.org/asdg

The download center for all the samples in this book (the Code Depot), plus dozens of other common ActionScript examples. Also provides book updates, technotes, sample chapters, errata, and news. Maintained by the author.

moockmarks

http://www.moock.org/moockmarks

The author's bookmarks. Lists hundreds of sites with technical information and design inspiration for Flash and web design in general.

Macromedia's Flash Support Center

http://www.macromedia.com/support/flash

A thriving collection of information for ActionScript and Flash. Contains the entire text of Macromedia's ActionScript Dictionary and a regularly updated database of technotes for Flash developers. Start your search for answers here. A trial version of the Flash authoring tool can be downloaded from: http://www.macromedia.com/software/flash/trial.

Macromedia Exchange for Flash

http://www.macromedia.com/exchange/flash

A collection of professional-quality components, *.fla* project files, and ActionScript samples. Content is hosted and endorsed by Macromedia but codeveloped by the Flash community. Access to samples requires use of an installation utility called the Extension Manager, which is also available at the Exchange.

Flash Kit

http://www.flashkit.com

Internet.com's Flash Kit is the largest third-party source of Flash help. The site has everything under the Flash sun: sample files, message boards, tutorials, news, feature articles, interviews, chat, gallery, and much more. Unfortunately, the incredible volume sometimes obscures worthwhile content. Of special interest to Flash programmers are the ActionScript sample *.fla* files available for download at http://www.flashkit.com/movies/Scripting.

Ultrashock

http://www.ultrashock.com

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

A general portal for Flash developers, Ultrashock maintains a lively collection of ActionScript-centric *.fla* files and tutorials, as well as a well-trafficked developer forum.

We're Here

http://www.were-here.com

A large, active group of web-based message boards for discussing Flash and ActionScript. Also features articles, example *.fla* files, tutorials, and Flash news.

Flashcoders mailing list

http://chattyfig.figleaf.com

Administered by Branden Hall, this active Flash developer's email list focuses specifically on intermediate to advanced ActionScript. It is a great place to learn practical techniques and have questions answered by peers, but it is *not* for general Flash questions or site checks. Archives are searchable and posted on a monthly basis. Check the archives for answers before posting questions.

Brandon Williams's and Ethan Kennedy's Flash Experiments

http://www20.brinkster.com/ahab/flash5

A large collection of advanced math-based ActionScript demos. Dozens of sample *.fla* files are available for study, covering 3D, fractals, and many visual physics effects. Includes in-depth articles on math-based programming in Flash.

Flash MX 3D component system

http://www.benchun.net/mx3d

A series of 3D engine components for Flash Player 6. Written by Ben Chun, MIT alumnus and Macromedia engineer.

Flash API

http://www.yestoall.com/flashAPI

An assortment of libraries for Flash development. Includes visual tools (such as skewing and spinning), object enhancements (such as sound pan control), and utilities (such as timers, loaders, easing equations, and distance calculators).

[PROTO]type

http://www.layer51.com/proto

A community-based website for posting extensions to ActionScript's built-in classes (in other words, new methods and properties for things such as *String*, *Array*, and *XML*). There are hundreds of submissions, many of which are very useful.

The *comp.graphics.algorithms* FAQ

http://www.faqs.org/faqs/graphics/algorithms-faq

Describes mathematical solutions to a wide variety of graphics programming problems. Ranges from simpler topics (such as, "How do I find the distance from a point to a line?") to advanced 3D issues, normally outside the realm of ActionScript programming.

Archived Macromedia Flash Players

http://www.macromedia.com/support/flash/ts/documents/oldplayers.htm

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Download old versions of the Macromedia Flash Player for testing.

General game-programming resources

The following sites offer useful information for designing, planning, and programming games (these are non-ActionScript-specific sites):

http://www.gamedev.net
http://www.flipcode.com
http://www.javascript-games.org

Flash and Active Server Pages (ASP)

The following articles from ASP 101 provide a good introduction to integrating Flash with ASP to create a simple Flash form:

http://www.asp101.com/articles/flash
http://www.asp101.com/articles/flash2

Macromedia Flash Communication Server MX (Comm Server)

The Comm Server API (e.g., *Remote SharedObject*, *Camera*, *Microphone*, *NetConnection*, and *NetStream*) is used to create multiuser web applications. Information about Comm Server's real-time data, video, and audio streaming for Flash can be found at:

http://www.macromedia.com/software/flashcom
http://www.flashcom.info

Macromedia Flash Remoting MX (Flash Remoting)

Flash Remoting allows you to use Flash as a front end to remote web applications and web services, including those built with ColdFusion MX. For more information on Flash Remoting, see:

http://www.macromedia.com/software/flash/flashremoting

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Appendix A. Resources

A.3 ECMA-262 Resources

Because ActionScript and JavaScript are both based on the ECMA-262 scripting language specification, they share a common core syntax. Hence, although the following resources do not focus specifically on ActionScript, they are worthwhile reading for ActionScript programmers:

ECMA-262 Language Specification

http://www.ecma.ch/ecma1/STAND/ECMA-262.HTM

The official ECMA-262 language specification from ECMA, the body that maintains the language upon which ActionScript is based. Netscape's JavaScript and Microsoft's Jscript are two other implementations of ECMA-262. This is a highly technical document used to create things such as a conforming ECMA interpreter.

Netscape's Core JavaScript Guide

http://developer.netscape.com/docs/manuals/js/core/jsguide/contents.htm

Netscape provides very readable documentation on JavaScript in this detailed explanation of JavaScript's core language features.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Appendix A. Resources

A.4 Object-Oriented Programming

Netscape's *Details of the Object Model*

http://developer.netscape.com/docs/manuals/js/core/jsguide/obj2.htm

An article explaining the prototype-based implementation of OOP, inheritance, and classes in JavaScript. Concepts are illustrated with numerous code samples. Netscape makes a special effort to explain JavaScript's OOP in terms that C and Java coders will understand.

Sun's Object-Oriented Programming Concepts (from The Java Tutorial)

http://java.sun.com/docs/books/tutorial/java/concepts

Although intended as a primer for aspiring Java programmers, this site covers the basics of OOP in generalized terms and makes good background reading for OOP in any language.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Appendix A. Resources

A.5 SWF File Format

Whereas the Flash Player is not open sourced, the *.swf* file format itself is publicly available (which is one reason it has garnered a lot of third-party support). For further information, here are some resources:

Macromedia Flash file format (SWF)

http://www.macromedia.com/software/flash/open/licensing/fileformat

Here you can obtain the SWF SDK from Macromedia, "a set of tools for developers to write Macromedia Flash (SWF) files, documentation of the Macromedia Flash file format (SWF), and code to write SWF files."

OpenSWF

http://www.openswf.org

A general information center for developers of SWF implementations. Includes tutorials, message boards, resources, and source code.

Ming

http://ming.sourceforge.net

An open source C library for generating SWF format movies, plus a set of wrappers for using the library in C++ and popular scripting languages such as PHP, Python, and Ruby. See http://www.opensource.org for information on open source software.

Perl :: Flash

http://www.2shortplanks.com/flash

A Perl library to allow the "dynamic, programmable generation of Flash/SWF movies." Created by Simon Wistow.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Appendix A. Resources

A.6 Character Encoding and Unicode

Unicode Character Ranges and the Unicode Fonts that Support Them

http://www.alanwood.net/unicode/fontsbyrange.html

An excellent place to find a font capable of displaying a particular range of Unicode characters.

Unicode and Multilingual Support in HTML, Fonts, Web Browsers and Other Applications

http://www.alanwood.net/unicode

Great information about using Unicode on the Web.

A Brief Introduction to Code Pages and Unicode

http://www-106.ibm.com/developerworks/library/codepages.html

An informative but digestible overview of the evolution of character encoding.

A tutorial on character code issues

http://www.cs.tut.fi/~jkorpela/chars.html

A highly informative collection of articles on the basics of character representation in computers. By Jukka Korpela.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Part III: Appendixes

Appendix B. Latin 1 Character Repertoire and Keycodes

Table B-1 lists the characters in the Latin 1 character repertoire, with Unicode equivalents in the range of `U+0000` to `U+00FF` (that is, C0 Controls, Basic Latin, C1 Controls, and Latin 1 Supplemental). The table's first column (labeled *Dec*), lists each character's code point in decimal (the standard ASCII or Latin 1 value), the second column provides the Unicode escape sequence for the character, and the third column describes or shows the character itself. See Chapter 4, for more information on character encoding in Flash. Table B-2 and Table B-3 list the keycodes used with the *Key* object. Although there is some overlap, keycodes represent physical keys on the keyboard and have no direct relation to the Latin 1 characters listed in Table B-1. That said, you'll notice that the keycodes for letters and numbers, listed in Table B-3, are identical to the Latin 1 code points (and ASCII values) for the letters A-Z and the numbers 0-9. See *Key.getCode( )* and the *Key* object in the ActionScript Language Reference for more information on keycodes.

For supplementary reading on the topic of character encoding, see the following resources:

A tutorial on character code issues

http://www.cs.tut.fi/~jkorpela/chars.html

A highly informative series of articles on character representation in computers. By Jukka Korpela.

Unicode FAQ

http://www.unicode.org/unicode/faq

A good question-and-answer format overview of Unicode, an international standard for character encoding.

To find the code point for a Unicode character not included in Latin 1, see:

Vendor Mappings

http://www.unicode.org/Public/MAPPINGS/VENDORS

Lists vendor-specific code pages with corresponding Unicode character code points. If you know the code page used for text entry on your computer, this will help you find the characters you can type.

Where is my character?

http://www.unicode.org/unicode/standard/where

A helpful document that suggests ways to track down a character in the Unicode character set.

Unicode Code Charts

http://www.unicode.org/charts

A collection of .pdf files that describe and visually depict the characters in Unicode.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Unihan Database

http://www.unicode.org/charts/unihan.html

The section of the Unicode Character Database that includes the code points for many Asian characters, including those in the Shift-JIS character set, Flash 5's supported set of Japanese characters.

Table B-1. ISO 8859-1 (Latin 1) characters and Unicode mappings

Dec	Unicode	Description	Dec	Unicode	Description
0	\u0000	[null]	19	\u0013	[device control three]
1	\u0001	[start of heading]	20	\u0014	[device control four]
2	\u0002	[start of text]	21	\u0015	[negative acknowledge]
3	\u0003	[end of text]	22	\u0016	[synchronous idle]
4	\u0004	[end of transmission]	23	\u0017	[end of transmission block]
5	\u0005	[enquiry]	24	\u0018	[cancel]
6	\u0006	[acknowledge]	25	\u0019	[end of medium]
7	\u0007	[bell]	26	\u001a	[substitute]
8	\u0008	[backspace]	27	\u001b	[escape]
9	\u0009	[horizontal tabulation]	28	\u001c	[file separator]
10	\u000a	[line feed]	29	\u001d	[group separator]
11	\u000b	[vertical tabulation]	30	\u001e	[record separator]
12	\u000c	[form feed]	31	\u001f	[unit separator]
13	\u000d	[carriage feed]	32	\u0020	[space]
14	\u000e	[shift out]	33	\u0021	!
15	\u000f	[shift in]	34	\u0022	" (straight quotes)
16	\u0010	[data link escape]	35	\u0023	#
17	\u0011	[device control one]	36	\u0024	$
18	\u0012	[device control two]	37	\u0025	%
38	\u0026	&	74	\u004a	J
39	\u0027	' (straight apostrophe)	75	\u004b	K
40	\u0028	(	76	\u004c	L
41	\u0029	)	77	\u004d	M
42	\u002a	*	78	\u004e	N
43	\u002b	+	79	\u004f	O
44	\u002c	, (comma)	80	\u0050	P
45	\u002d	- (minus)	81	\u0051	Q
46	\u002e	. (period)	82	\u0052	R
47	\u002f	/	83	\u0053	S
48	\u0030	0	84	\u0054	T
49	\u0031	1	85	\u0055	U
50	\u0032	2	86	\u0056	V

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

51	\u0033	3	87	\u0057	W
52	\u0034	4	88	\u0058	X
53	\u0035	5	89	\u0059	Y
54	\u0036	6	90	\u005a	Z
55	\u0037	7	91	\u005b	[
56	\u0038	8	92	\u005c	\
57	\u0039	9	93	\u005d	]
58	\u003a	:	94	\u005e	^
59	\u003b	;	95	\u005f	_ (underscore)
60	\u003c	<	96	\u0060	` (accent grave)
61	\u003d	=	97	\u0061	a
62	\u003e	>	98	\u0062	b
63	\u003f	?	99	\u0063	c
64	\u0040	@	100	\u0064	d
65	\u0041	A	101	\u0065	e
66	\u0042	B	102	\u0066	f
67	\u0043	C	103	\u0067	g
68	\u0044	D	104	\u0068	h
69	\u0045	E	105	\u0069	i
70	\u0046	F	106	\u006a	j
71	\u0047	G	107	\u006b	k
72	\u0048	H	108	\u006c	l
73	\u0049	I	109	\u006d	m
110	\u006e	n	146	\u0092	control chr
111	\u006f	o	147	\u0093	control chr
112	\u0070	p	148	\u0094	control chr
113	\u0071	q	149	\u0095	control chr
114	\u0072	r	150	\u0096	control chr
115	\u0073	s	151	\u0097	control chr
116	\u0074	t	152	\u0098	control chr
117	\u0075	u	153	\u0099	control chr
118	\u0076	v	154	\u009a	control chr
119	\u0077	w	155	\u009b	control chr
120	\u0078	x	156	\u009c	control chr
121	\u0079	y	157	\u009d	control chr
122	\u007a	z	158	\u009e	control chr
123	\u007b	{	159	\u009f	control chr
124	\u007c	|	160	\u00a0	[no break space]
125	\u007d	}	161	\u00a1	?
126	\u007e	~	162	\u00a2	?
127	\u007f	[delete]	163	\u00a3	?

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

128	\u0080	control chr	164	\u00a4	¤
129	\u0081	control chr	165	\u00a5	?
130	\u0082	control chr	166	\u00a6	
131	\u0083	control chr	167	\u00a7	§
132	\u0084	control chr	168	\u00a8	?
133	\u0085	control chr	169	\u00a9	©
134	\u0086	control chr	170	\u00aa	?
135	\u0087	control chr	171	\u00ab	«
136	\u0088	control chr	172	\u00ac	¬
137	\u0089	control chr	173	\u00ad	-
138	\u008a	control chr	174	\u00ae	®
139	\u008b	control chr	175	\u00af	?
140	\u008c	control chr	176	\u00b0	°
141	\u008d	control chr	177	\u00b1	±
142	\u008e	control chr	178	\u00b2	^{2}
143	\u008f	control chr	179	\u00b3	^{3}
144	\u0090	control chr	180	\u00b4	?
145	\u0091	control chr	181	\u00b5	µ
182	\u00b6	¶	219	\u00db	u
183	\u00b7		220	\u00dc	U
184	\u00b8	?	221	\u00dd	Y
185	\u00b9	^{1}	222	\u00de	
186	\u00ba	?	223	\u00df	?
187	\u00bb	»	224	\u00e0	a
188	\u00bc	$^1/_4$	225	\u00e1	a
189	\u00bd	$^1/_2$	226	\u00e2	a
190	\u00be	$^3/_4$	227	\u00e3	a
191	\u00bf	?	228	\u00e4	a
192	\u00c0	A	229	\u00e5	a
193	\u00c1	A	230	\u00e6	?
194	\u00c2	?	231	\u00e7	c
195	\u00c3	A	232	\u00e8	e
196	\u00c4	A	233	\u00e9	e
197	\u00c5	A	234	\u00ea	e
198	\u00c6	?	235	\u00eb	e
199	\u00c7	C	236	\u00ec	i
200	\u00c8	E	237	\u00ed	i
201	\u00c9	E	238	\u00ee	i
202	\u00ca	E	239	\u00ef	i
203	\u00cb	E	240	\u00f0	

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

204	\u00cc	I	241	\u00f1	n
205	\u00cd	I	242	\u00f2	o
206	\u00ce	I	243	\u00f3	o
207	\u00cf	I	244	\u00f4	o
208	\u00d0	Ð	245	\u00f5	o
209	\u00d1	N	246	\u00f6	o
210	\u00d2	O	247	\u00f7	?
211	\u00d3	O	248	\u00f8	
212	\u00d4	O	249	\u00f9	u
213	\u00d5	O	250	\u00fa	u
214	\u00d6	O	251	\u00fb	U
215	\u00d7	x	252	\u00fc	u
216	\u00d8	O	253	\u00fd	
217	\u00d9	U	254	\u00fe	
218	\u00da	U	255	\u00ff	y

Table B-2 lists the keycodes for selected special keys on the keyboard. These keycodes are used only with the *Key* object. See Table 18-8 (under the *Key* object in the ActionScript Language Reference) for an alphabetical listing of the special properties.

Table B-2. Key object special keycodes

Key	Keycode	Property	Key	Keycode
Backspace	8	Key.BACKSPACE	Num Pad 9	105
Tab	9	Key.TAB	Num Pad *	106
Enter	13	Key.ENTER	Num Pad +	107
Shift	16	Key.SHIFT	Num Pad Enter	13 (not trappable as 108)
Control	17	Key.CONTROL	Num Pad -	109
Alt	18 (not trappable)		Num Pad .	110
Pause/Break	19		Num Pad /	111
Caps Lock	20	Key.CAPSLOCK	F1	112
Escape	27	Key.ESCAPE	F2	113
Space	32	Key.SPACE	F3	114
Page Up	33	Key.PGUP	F4	115
Page Down	34	Key.PGDN	F5	116
End	35	Key.END	F6	117
Home	36	Key.HOME	F7	118
Left arrow	37	Key.LEFT	F8	119
Up Arrow	38	Key.UP	F9	120
Right Arrow	39	Key.RIGHT	F10	121 (not trappable)

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Down arrow	40	Key.DOWN	F11	122
Print Screen	44		F12	123
Insert	45	Key.INSERT	Num Lock	144
Delete	46	Key.DELETEKEY	Scroll Lock	145
Num Pad 0	96		; :	186
Num Pad 1	97		+ =	187
Num Pad 2	98		- _	189
Num Pad 3	99		/ ?	191
Num Pad 4	100		` ~	192
Num Pad 5	101		[{	219
Num Pad 6	102		\ |	220
Num Pad 7	103		] }	221
Num Pad 8	104		" '	222

Table B-3 lists the keycodes for the letter and number keys on a keyboard. These keycodes are used only with the *Key* object (see *Key.getCode( )* and *Key.isDown( )* in the ActionScript Language Reference).

Table B-3. Key object letter and number keycodes

Key	Keycode		Key	Keycode		Key	Keycode
A	65		M	77		Y	89
B	66		N	78		Z	90
C	67		O	79		0	48
D	68		P	80		1	49
E	69		Q	81		2	50
F	70		R	82		3	51
G	71		S	83		4	52
H	72		T	84		5	53
I	73		U	85		6	54
J	74		V	86		7	55
K	75		W	87		8	56
L	76		X	88		9	57

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Part III: Appendixes

Appendix C. Backward Compatibility and Player Build Updates

Flash allows you to publish *.swf* files in formats that are compatible with previous versions of the Player (via File → Publish Settings → Flash → Version). By the time you read this, the majority of users will have at least version 6 of the Flash Player, but you may prefer to author for Flash Player 5 or 4 instead, such as when deploying to handheld devices that don't yet support Flash Player 6. Even if you are authoring exclusively for Flash 6, this appendix will help you avoid using deprecated ActionScript and bring you up to speed on the preferred methods in Flash 6. Furthermore, it will help you understand or upgrade legacy code that you may encounter.

See `capabilities.version`, *getVersion( )*, and `$version` in the ActionScript Language Reference for details on detecting the version of the Flash Player at runtime. See http://www.moock.org/webdesign/flash/detection/moockfpi for a version detection script that runs in the browser.

When authoring for Flash 4 or 5, use the *deprecated* techniques (i.e., those that are outdated, but supported for backward compatibility) listed in Table C-1 (under Section C.3 in this appendix). See also "Using Flash MX ActionScript with older versions of Flash" under "Understanding the ActionScript Language" in the electronic help files under Help Using Flash. Deprecated features of ActionScript are listed in the Actions panel Toolbox under the Deprecated folder.

For statistics on the distribution of various Flash Player versions, see:

http://www.macromedia.com/software/flash/survey/whitepaper

To download archived versions of the Flash Player, visit the following pages:

http://www.macromedia.com/support/flash/downloads.html
http://www.macromedia.com/support/flash/ts/documents/oldplayers.htm

See the *Capabilities* object in the ActionScript Language Reference for information on authoring Flash content for devices.

To run scripts in Flash Player 4, you must set the Version of your *.swf* file to Flash 4 under the Flash tab under File Publish Settings. All code is disabled if you attempt to run a Flash 5 or later *.swf* file in Flash Player 4. To run scripts in Flash Player 5, you must set the Version of your *.swf* file to Flash 5 under the Flash tab under File Publish Settings. All code is disabled if you attempt to run a Flash 6 or later *.swf* file in Flash Player 5.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

About Safari | Terms of Service | Privacy Policy | Contact Us | Help | Submit a Problem

Copyright © 2002 O'Reilly & Associates, Inc. All rights reserved.

1005 Gravenstein Highway North
Sebastopol, CA 95472

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Appendix C. Backward Compatibility and Player Build Updates

C.1 Best Practices in Flash 6 ActionScript

Very little of Flash 5 ActionScript was formally deprecated in Flash 6. The following formal changes to the language were introduced in Flash 6:

- The *eval( )* function cannot be used on the left side of an assignment operation.
- Function scope is defined permanently at declaration time instead of at invocation time. See Section 9.7.

The following list presents new features that are preferred over previous Flash 5 alternatives:

- Global variables should be defined on `_global`, not `_root` or *Object*.
- Superclass constructors should be executed via the *super* operator, not a manual reference to the superclass constructor function.
- All imported text should be in Unicode, not Latin 1 or Shift-JIS.
- Movie clip and button event handlers should be assigned as object properties, not with the *onClipEvent(`event`)* or *on(`event`)* syntax. See the ActionScript Language Reference for specific exceptions for individual events.
- Text field content must be accessed via the *`TextField`*`.text` property, not the text field's variable name.
- JPEG files should be loaded directly using *loadMovie( )*, not wrapped in *.swf* files.
- Often, timeline and *onEnterFrame( )* loops can be replaced with *setInterval( )*.
- A local *SharedObject* should be used in favor of a JavaScript cookie.
- `System.capabilities.version` is preferred over *getVersion( )* (though *getVersion( )* must still be used in movies displayed in Flash Player 5).
- The *Key* and *Mouse* object events—*onKeyDown( )*, *onKeyUp( )*, *onMouseDown( )*, and *onMouseUp( )*—should be used in place of analogous *MovieClip* events, except when *updateAfterEvent( )* is required.
- The *LoadVars* class supercedes the *loadVariables( )* function.
- `Stage.scaleMode` and `Stage.showMenu` replace *fscommand("allowscale")* and *fscommand("showmenu")*.

For more best practices in Flash 6 ActionScript, see:

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

http://www.macromedia.com/desdev/mx/flash/whitepapers/actionscript_standards.pdf

As a rule, the new objects, classes, statements, and operators introduced in Flash 6 (as listed in the Preface) cannot be used when exporting to Flash 5 format. Three exceptions are the *switch* statement and the strict equality and inequality operators (`= = =` and `!==`), which are all available when exporting from the Flash MX authoring tool to Flash 5 format. When a movie is set to export as version 5, the Flash MX Output window reports any incompatibility warnings at export time.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Appendix C. Backward Compatibility and Player Build Updates

C.2 Player Build Updates

The following sections describe changes and fixes introduced by minor versions of the Flash Player.

C.2.1 Updates to Flash Player 6, Build 61

As this book was going to press (November 2002), Macromedia released Build 61 of Flash Player 6 (Flash Player 6.0.61.0). The following list summarizes the main features of the new build:

Windowless mode

Implemented for Netscape for Windows and Mac OS X, in addition to Internet Explorer for Windows. See the `WMODE` attribute in Appendix H.

Improved ActionScript performance

Variable registers are now used to speed up variable access, which generally results in faster ActionScript execution times. The improved performance is particularly noticeable in FUI components.

Customization of HTTP headers

HTTP headers for POST operations can be customized using the new *XML.addRequestHeader( )* and *LoadVars.addRequestHeader( )* methods.

Dependent runtime shared libraries

Allows runtime shared libraries to use items from other runtime shared libraries.

Dynamic Accessibility properties (Windows, ActiveX only)

Allows properties of accessible objects (previously settable via the Flash MX Accessibility panel at authoring time only) to be set at runtime using the new *Accessibility.updateProperties( )* method and the `Accessibility._accProps` object (whose properties are `silent`, `forceSimple`, `name`, description, and `shortcut`).

Linux support

Support for Mozilla 1.1 and the Standalone Player under RedHat Linux 7.3.

Miscellaneous fixes

Fixes were made, especially to Flash Player 6 for Internet Explorer for Windows.

For many more details, see the release notes at:

http://www.macromedia.com/support/flash/releasenotes/player/rn_6.html

C.2.2 Updates to Flash Player 6, Build 47

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

The following list summarizes the major changes that were made between Build 40 and Build 47 of Flash Player 6 (Flash Player 6.0.47.0), released in August 2002:

- When minimized or hidden, Netscape for Windows no longer consumes CPU cycles.
- Multiple cursors no longer occasionally appear in text fields.
- Fixed security issue — "Macromedia Flash URL Modification Issue." See:

  http://www.macromedia.com/v1/handlers/index.cfm?ID=23294

C.2.3 Updates to Flash Player 6, Build 40

The following list summarizes the major changes that were made between Build 29 and Build 40 of Flash Player 6 (Flash Player 6.0.40.0), released in July 2002. Various minor optimizations and bug fixes are discussed at: http://www.macromedia.com/support/flash/ts/documents/flashplayer_r40.htm

- POST method fixed in Netscape (all platforms) and Internet Explorer for Macintosh.
- Plugin for Netscape 6.2 and above now supports JavaScript communication and *fscommand( )* via XPConnect (Netscape's replacement for its LiveConnect technology).
- Fixed word-wrap bug that made text appear to quiver.
- *Sound.start( )* enabled for stream sounds. See *Sound.start( )* for details.
- Device-font masking feature added. See *MovieClip.setMask( )* for details.
- MP3 ID3 properties now accessible via *Sound* objects, as described under `Sound.id3` in the ActionScript Language Reference.
- Windows German keyboard input fixes (German keyboards can now type @ and other special characters).
- Fixed high-ASCII-to-Unicode character mapping on Macintosh. High-ASCII should now display correctly on the Macintosh Flash Player.
- A new `OBJECT/EMBED` attribute, `allowScriptAccess`, can prohibit unwanted cross-domain scripting of an HTML page. See http://www.macromedia.com/support/flash/ts/documents/allow_script_access.htm.

C.2.4 Updates to Flash Player 6, Build 29

Build 29 of Flash Player 6 (Flash Player 6.0.29.0), released in May 2002, fixed the following bugs that were present in the original public version of Flash Player 6 (Flash Player 6.0.25.0, released in March 2002):

- Halting the download of loading media (*.swf*, *.jpg*, or *.mp3* files) now stops the download of the file properly.
- An event sound attached to a *Sound* object now stops playing when there are no more references to the

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Sound object.

- Large media (video or MP3 content) no longer causes the Player to lock up.

C.2.5 Updates to Flash Player 5, Build 41 and 42

The following list summarizes the major changes that were made in Build 41 (Netscape) and 42 (Internet Explorer) of Flash Player 5 (the previous public build was 30, initially released with the Flash 5 authoring tool in August 2000):

- I-beams in text fields take on the color of the text.
- Movies in a table cell do not cause Internet Explorer 5.5 to crash.
- A text field's scroll position does not reset when field's content is modified.
- Text fields with embedded fonts do not leak outside their visual boundary when scrolled.
- The *XML*`.contentType` property was added.
- The *XML*`.ignoreWhite` property was added.
- When XML source is parsed and the characters `&`, `'`, `"`, `<`, and `>` appear in a text node, they are converted to the following entities: `&`, `'`, `"`, `<`, and `>`. This conversion is transparent in Flash because the entities are converted back to characters when an XML object is converted to a string; however, the entities *will* show up in XML source sent to the server.
- *Math.random( )* no longer returns the value 1. The maximum return value is 0.999...
- General performance is improved, especially on Windows 98.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Appendix C. Backward Compatibility and Player Build Updates

C.3 Flash 4 Versus Flash 5 (and Later)

Table C-1 summarizes key backward-compatibility issues and differences between ActionScript in Flash 4 versus Flash 5 and Flash 6.

Table C-1. Flash 4 backward-compatibility issues

Topic	Description
Creating variables	Flash 4's *set* function has been replaced by the *var* statement.
Variable and timeline references	The Flash 4-style slash-colon constructions (`/square:area`) have been superceded by dot notation (`square.area`).
String comparison operators	The Flash 4 string comparison operators—`eq`, `ne`, `ge`, `gt`, `le`, `lt`—have been superceded by the following operators in Flash 5 and later: `= =`, `!=`, `>=`, `>`, `<=`, `<`.
String concatenation operator	When creating Flash 4 content in Flash 5 or later, use the *add* operator instead of Flash 4's `&` operator. When authoring for Flash 5 or later, use the `+` operator for string concatenation.
String length	Flash 4's *length( )* function (e.g., *length(`myString`)*) has been superceded by the *`String`*`.length` property (e.g., *`myString`*`.length`).
Substring extraction	Flash 4's *substring( )* function (e.g., *substring(`myString`, 1, 3)*) has been superceded by the *substring( )*, *substr( )*, and *slice( )* methods. Note that *substring( )* differs in Flash 4 and Flash 5.
Character code point functions	Flash 4's *chr( )* and *mbchr( )* functions (used to create a character from a code point) have been superceded by *String.fromCharCode( )*. Flash 4's *ord( )* and *mbord( )* functions (used to determine the code point of a character) have been superceded by *String.charCodeAt( )*.
Datatype conversion	When importing Flash 4 files, Flash 5 and Flash MX automatically insert the *Number( )* function around any numeric data that is used as an operand of the following potentially ambiguous operators: `+`, `= =`, `!=`, `<>`, `<`, `>`, `>=`, `<=`.
The *ifFrameLoaded* statement	Flash 3's *ifFrameLoaded* statement is deprecated. Use the `_totalframes` and `_framesloaded` *MovieClip* properties to create preloading code.
Infinite loops	Flash 4 allowed a maximum of 200,000 loop iterations. Flash 5 and Flash 6 allow 15 seconds for loops, after which they warn users that the movie has stopped responding. See Section 8.6.3.
Subroutines versus functions	In Flash 4, a subroutine could be created by attaching a block of code to a frame with a label and executing it using the *call( )* statement. In Flash 5 and later, functions replace Flash 4 subroutines.
Clip events	Flash 4 supported only button events (i.e., functions starting with *on( )*). For modern event-handling techniques, see Chapter 10.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Capturing keystrokes	In Flash 4, *keyPress* was the only means of capturing keystrokes. In Flash 5 and later, the *Key* object and the movie clip events *onClipEvent(keyDown)* and *onClipEvent(keyUp)* offer much greater control over keyboard interaction.
Tell Target deprecated	Flash 4's *Tell Target* action (used to control remote movie clips) is replaced by properties and methods accessed using dot notation and the *with* statement. See "Whither Tell Target" later in this appendix.
Get Property deprecated	Flash 4's *Get Property* command is no longer required for movie clip property access. Use the dot operator instead. See Chapter 13.
int deprecated	Flash 4's *int( )* function (used to truncate floats to integers) has been superceded by *Math.floor( )*, *Math.ceil( )*, and *Math.round( )*.
Random number generation	Flash 4's *random( )* function (used to generate a random number) has been superceded by *Math.random( )*.
Toggle High Quality deprecated	Flash 4's *Toggle High Quality* action (used to set the rendering quality of the player) has been superceded by the global `_quality` property.
`_highquality` deprecated	Flash 4's `_highquality` property has been superceded by the global `_quality` property.
Math object support in Flash 4	The functions and properties of the *Math* object (e.g., *Math.cos( )*, `Math.PI`) are not natively supported by Flash Player 4. The values, however, are approximated when a movie is exported in Flash 4 format.
loadMovie versus *loadMovieNum*	Flash 3's *loadMovie( )* onto a numbered level is superceded by *loadMovieNum( )* (which accepts an integer level argument) as of Flash 5. Flash 4's *loadMovie( )* into a target movie clip is still available as *loadMovie( )* in Flash 5 and later.
Printing	Flash 5 and later support native print functions, which are available in Flash 4.0.20.0 and later as a modified *Get URL* action only.
Objects and classes not supported	Flash 4 does not support any of Flash 5's or Flash 6's built-in objects and classes (with the exception of *Math* as noted earlier in this table).

C.3.1 Controlling Movie Clips

In Chapter 13, we discussed how to control clips using Flash 5 (and later) techniques. Here we consider the equivalent Flash 4 techniques.

Prior to Flash 5, we executed special *Movie Clip Actions* to control a movie clip. We said, "Tell the clip named `eyes` to play," using the following:

```
Begin Tell Target ("eyes")
 Play
End Tell Target
```

But as of Flash 5, movie clips can be controlled more directly, through built-in methods. For example:

```
eyes.play();
```

Similarly, to access the built-in properties of a movie clip prior to Flash 5, we used explicit property-getting and property-setting commands, such as:

```
GetProperty ("ball", _width)
Set Property ("ball", X Scale) = 90
```

As of Flash 5, we can retrieve and set a movie clip's properties using the dot operator, just as we access the

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

properties of any object:

```
ball._width;
ball._xscale = 90;
```

Prior to Flash 5, to access variables inside a movie clip, we used a colon to separate the clip name from the variable name:

```
Set Variable: "x" = myClip:myVariable
```

As of Flash 5, a variable in a movie clip is simply a property of that clip object, so we use the dot operator to set and retrieve variable values:

```
x = myClip.myVariable;
```

Finally, prior to Flash 5, we accessed nested levels of movie clips using a directory-tree metaphor of slashes and dots:

```
clipA/clipB/clipC
../../../clipC
```

Because movie clips are object-like data as of Flash 5, we can access one clip as a property of another clip. We therefore use the dot operator to access so-called *nested* clips, and we use a clip's reserved `_parent` property to refer to the clip that contains it:

```
clipA.clipB.clipC;
_parent._parent._parent.clipC;
```

C.3.2 Flash 4 Versus Flash 5 Variable Access Syntax

The Flash 4-style slash-colon constructions, such as `/square:area`, have been superceded by dot syntax, a much more convenient way to refer to variables and timelines, in Flash 5 and later. The old syntax is deprecated and no longer recommended. Table C-2 shows equivalencies between Flash 4 and Flash 5 and later syntax when addressing variables. See Table C-1 for other syntactical differences.

Table C-2. Flash 4 versus Flash 5 and later variable addressing syntax

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Flash 4 syntax	Flash 5 and later syntax	Refers to . . .
`/`	`_root`	Movie's main timeline
`/:x`	`_root.x`	Variable `x` on movie's main timeline
`/clip1:x`	`_root.clip1.x`	Variable `x` in instance `clip1` on movie's main timeline
`/clip1/clip2:x`	`_root.clip1.clip2.x`	Variable `x` in instance `clip2`, within instance `clip1`, within the main movie timeline
`../`	`_parent`	Timeline upon which the current clip resides (one level up from current clip timeline[C])
`../:x`	`_parent.x`	Variable `x` on timeline upon which the current clip resides (one level up from current clip timeline)
`../../:x`	`_parent._parent.x`	Variable `x` on timeline that contains the clip that contains the current clip (two levels up from current clip timeline)
`clip1:x`	`clip1.x`	Variable `x` in instance `clip1`, where `clip1` resides on the current timeline
`clip1/clip2:x`	`clip1.clip2.x`	Variable `x` in instance `clip2`, where `clip2` resides within `clip1`, which in turn resides on current timeline
`_level1:x`	`_level1.x`	Variable `x` on the main timeline of a movie loaded onto level 1
`_level2:x`	`_level2.x`	Variable `x` on the main timeline of a movie loaded onto level 2

[C] The "current clip timeline" is the timeline that contains the code with the variable reference.

C.3.3 Local Variables in Flash 4-Style Subroutines

Although functions are the preferred mechanism for producing portable code modules, Flash 5 and Flash 6 still support Flash 4-style *subroutines*. In Flash 4, a subroutine could be created by attaching a block of code to a frame with a label. Later, the subroutine could be executed remotely via the *Call* action. But in Flash 4, any variable declared in a subroutine was nonlocal and persisted for the lifetime of the timeline on which it was defined. In Flash 5 and later, you can create local variables in subroutines the same way we create them in functions—using the *var* statement. However, variables defined with *var* in a subroutine are created as local variables only when the subroutine is executed via the *Call* function. If the script on the subroutine frame is executed as a result of the playhead simply entering the frame, the *var* statement declares a normal timeline nonlocal variable. Regardless, the more modern functions and local function variables should be used instead of subroutines.

C.3.4 Flash 4-to-Flash 5 Datatype Conversion

In Flash 4, the string operators and the numeric operators were completely distinct—one set of operators worked only with strings, and the other set worked only with numbers. For example, the string concatenation operator in Flash 4 was `&`, but the mathematical addition operator was `+`. Similarly, string comparisons were performed with the `eq` and `ne` operators, but numeric comparisons were accomplished via `=` and `<>`. Table C-3 lists the Flash 5 and later syntax for analogous Flash 4 operators.

Table C-3. Flash 4 versus Flash 5 operators

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Operation	Flash 4 syntax	Flash 5 and later syntax
String concatenation	`&`	`+` or *add*
String equality	`eq`	`= =`
String inequality	`ne`	`!=`
String comparison	`ge`, `gt`, `le`, `lt`	`>=`, `>`, `<=`, `<`
Numeric addition	`+`	`+`
Numeric equality	`=`	`= =`
Numeric inequality	`<>`	`!=`
Numeric comparison	`>=`, `>`, `<=`, `<`	`>=`, `>`, `<=`, `<`

Some Flash 5 and later operators can operate on both strings and numbers. For example, when the `+` operator is used with strings, it concatenates its operands to form a new string. But when it is used with numbers, the `+` operator adds its two operands mathematically. Similarly, in Flash 5 and later, the equality operator (`= =`) and inequality operator (`!=`) are used to compare strings, numbers, and other datatypes.

Because many Flash 5 and later operators work with multiple datatypes, but Flash 4 operators do not, an ambiguity arises when a Flash 4 file is imported into Flash 5 or later. Therefore, when importing Flash 4 files, Flash 5 and Flash MX automatically insert the *Number( )* function around any numeric data that is used as an operand of the following potentially ambiguous operators (unless the operand is a numeric literal): `+`, `= =`, `!=`, `<>`, `<`, `>`, `>=`, and `<=`.

Flash 4 files converted to Flash 5 or Flash MX format will also have the string concatenation operator (`&`) changed to the new *add* operator. Table C-4 contains examples of Flash 4-to-Flash 5 operator translation.

Table C-4. Sample Flash 4-to-Flash 5 operator translations

Flash 4 syntax	Flash 5 and later syntax
`Loop While (count <= numRecords)`	`while (Number(count)<= Number(numRecords))`
`If (x = 15)`	`if(Number(x) = = 15)`
`If (y <> 20)`	`if(Number(y) != 20)`
`Set Variable: "lastName" = "kavanagh"`	`lastName = "kavanagh"`
`Set Variable: "name" = "molly" & lastName`	`name = "molly" add lastName`

C.3.5 Flash 4 Events

Most ActionScript events were first introduced in Flash 5. If you're exporting to Flash 4 format, use the button event handlers only (only button events were supported in Flash 4), and test your work carefully in Flash Player 4. To determine whether an event is supported by a particular version of the Flash Player, see the appropriate event entry in the *Language Reference.*

C.3.6 Flash 4 Versus Flash 5 String Operators

The Flash 4 string concatenation operator (`&`) performs a *different* operation (i.e., bitwise *AND*) in Flash 5 and later. If you're exporting a Flash 4 *.swf* file from a newer version of Flash, you must use the *add* operator to concatenate strings. Note that *add* is supported only for backward compatibility; the `+` operator is preferred in

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Flash 5 and later.

If you're exporting a Flash 4 *.swf* file, you must use the older `eq` and `ne` operators for string equality (`==`) and inequality (`!=`) comparisons. The `==` and `!=` operators are preferred in Flash 5 and later.

If you're exporting a Flash 4 *.swf* file, you must use the older `gt`, `ge`, `lt`, and `le` string comparison operators. The `>`, `>=`, `<`, and `<=` operators are the preferred equivalents in Flash 5 and later.

If you're exporting a Flash 4 *.swf* file, you must use the older *length( )* function to determine the length of a string. The *length( )* function is supported for backward compatibility, but the `String.length` property is preferred in Flash 5 and later. Here we use the Flash 4 *length( )* function to display the number of characters in the word "obsolete":

```
trace (length("obsolete"));  // Displays: 8
```

In Flash 5, the *String.substr( )* method is the string-extraction function that most closely resembles Flash 4's *substring( )* function, which also used a start index and a length to retrieve a substring. Flash 5's *String.substring( )* function uses start and end indexes.

If you're exporting a Flash 4 *.swf* file, you must use the older Flash 4 character-creation functions, *chr( )* and *mbchr( )*, to create characters from numeric codes. In Flash 5 and later, *String.fromCharCode( )* is preferred.

If you're exporting a Flash 4 *.swf* file, you must use the older Flash 4 code point functions, *ord( )* and *mbord( )* to determine the code point of a character. In Flash 5 and later, *String.charCodeAt( )* is preferred.

Because all the Flash 5 string operations and functions work with multibyte characters, there's no way in Flash 5 to force a single-byte operation as there was in Flash 4. For example, the *fromCharCode( )* function is as close as things get in Flash 5 to *chr( )*. The same is true of *mblength( )* and `length`, *mbsubstring( )* and *substr( )*, and *mbord( )* and *charCodeAt( )*.

C.3.7 Whither Tell Target?

In Flash 4, *Tell Target* was our main tool for referring to movie clips. *Tell Target*, bless its soul, was an unwieldy tool and is rendered obsolete by the much more elegant object model introduced in Flash 5. The *Tell Target* function has been deprecated (i.e., retired from recommended use). Although we can still use the *tellTarget( )* function to code in a Flash 4 manner, *tellTarget( )* may disappear in the future.

Consider the following code, which uses *Tell Target* to play an instance named `closingSequence`:

```
Begin Tell Target ("closingSequence")
  Play
End Tell Target
```

As of Flash 5, we simply invoke the much more convenient and readable *play( )* method on the `closingSequence` instance:

```
closingSequence.play();
```

Tell Target could also perform multiple operations on an instance within a code block, like so:

```
Begin Tell Target ("ball")
  (Set Property: ("ball", x Scale) = "5")
  Play
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

```
End Tell Target
```

As of Flash 5, the *with( )* statement, described in Chapter 6, is the preferred way to achieve similar results:

```
with (ball) {
  _xscale = 5;
  play();
}
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Part III: Appendixes

Appendix D. Differences from ECMA-262 and JavaScript

Naturally, this book focuses on the intricacies of ActionScript, but if you own O'Reilly's well-known *JavaScript: The Definitive Guide*, you'll notice that the reference section of the two books share many similarities.

Although ActionScript, like JavaScript, is based on the ECMA-262 standard, certain differences were necessitated by the constraints of Player size and backward compatibility. If you're porting code from JavaScript, JScript, or another ECMA-262-based language, you'll find Table D-1 valuable. It summarizes the intentional differences between ECMA-262, JavaScript, and ActionScript (for Flash 5 and later). Likewise, if you're porting ActionScript to another language, you'll be better able to avoid the pitfalls caused by ActionScript's deviation from the ECMA-262 standard.

Table D-1 reflects the *intentional* differences between Flash ActionScript and the ECMA-262 standard. It does not reflect any bugs that may exist in its attempted implementation of the standard. Furthermore, Table D-1 reflects ActionScript's behavior when the `#strict` pragma is not being used. (The undocumented, unsupported `#strict` pragma was intended to enhance ActionScript's conformance with the ECMA-262 standard. However, the `#strict` pragma should *not* be used, because it crashes Internet Explorer for Windows. See the entry for the `#strict` pragma in the *Language Reference* for details.)

Table D-1. Differences between ECMA-262, JavaScript, and ActionScript

Topic	Description
Exception handling	ActionScript does not support the exception handling statements *try*/*catch*/*finally* or *throw*.
escape( ) encoding (Flash 6 only)	In Flash 6, for characters between code point 127 and 255 (high ASCII), ActionScript's *escape( )* function encodes UTF-8 sequences using `%xx` rather than the method defined by the ECMA-262 specification. See *escape( )* in the ActionScript Language Reference.
Statement labels	ActionScript does not support labeled statements, used by the *break* and *continue* statements to control program flow.
String-to-Boolean conversion	In ECMA-262, all nonempty strings convert to `true`. In ActionScript, only the strings that can be converted to a valid nonzero number convert to `true`.
Case sensitivity	The ECMA-262 specification demands complete case sensitivity. In ActionScript, keywords are case-sensitive but identifiers are not. See Chapter 15, especially Table 15-1.
Function scope (Flash 5 only)	In Flash 5 (but not Flash 6), when a function from one timeline is assigned to a variable in a different movie clip's timeline, the assigned function's scope chain changes to that variable's timeline. In ECMA-262 and Flash 6 ActionScript, it's impossible to modify a function's scope chain through assignment; scope is determined permanently by the location of the function declaration statement.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Regular expressions	ActionScript does not support regular expressions. However, developer Pavils Jurjans has made a custom *RegExp* object available at http://www.jurjans.lv/flash/RegExp.html.
Event handler properties (Flash 5 only)	In Flash 5 (but not Flash 6) movie clip event handlers must be defined using *onClipEvent (*`eventName`*)*, and button event handlers must be defined using *on (*`eventName`*)*. In Flash 6, clip and button event handlers can be assigned as object properties, as in JavaScript. See Section 10.5 in Chapter 10.
Global variables (Flash 5 only)	Flash 5 ActionScript does not support true document-wide global variables. Global variables can be simulated by attaching properties to `Object.prototype`, as described under Section 12.5.3.5. In Flash 6, global variables are created on the `_global` object (which itself is not part of ECMA-262).
The *eval( )* function	ActionScript's *eval( )* function supports a small subset of ECMA-262's intended functionality; it works only when its argument is an identifier and is used only to dynamically generate references to identifiers. In Flash 6, it can't be used on the left side of an assignment statement.
`undefined` datatype conversion	In ActionScript, the special `undefined` value converts to the empty string ("") when used in a string context and converts to the number 0 when used in a numeric context. In ECMA-262, `undefined` converts to the string "undefined" in string contexts and to the numeric value `NaN` in numeric contexts.
The *Function* constructor	ActionScript does not support the *Function* constructor, which is used in JavaScript to create functions with the syntax *new Function ( );*.
Date object creation	ActionScript will not accept (i.e., will not parse) a human-readable date string, such as "January 9, 2001", when creating a new *Date* object.
switch statement (Flash 5 only)	Flash 5 ActionScript does not support the *switch*/*case*/*default* statements (used to phrase complex conditionals). The *switch* statement is supported in Flash 6 and can be used when exporting from the Flash MX authoring tool to a Flash 5 or Flash 4 format *.swf* file.
Language support (Flash 5 only)	ECMA-262 requires support of the Unicode character-encoding standard, which Flash 5 ActionScript does not support. Flash 5 uses the Latin 1 and Shift-JIS character sets and implements a subset of Unicode-style functions and conventions (such as `\u` escape sequences). Flash 6 fully supports Unicode.
Object model	Naturally, JavaScript includes built-in classes and objects that relate to web browsers, whereas Flash includes those that relate to Flash movies. For JavaScript programmers who are used to working with DHTML, it may be helpful to think of the main movie of a Flash document as analogous to an HTML document object and movie clips as analogous to layer objects.
Timed code execution (Flash 5 only)	In Flash 5, the *setTimeout( )* and *setInterval( )* methods of the JavaScript window object are not available in ActionScript but can be simulated with timeline and clip event loops, as discussed in Chapter 8. Flash 6 supports *setInterval( )*, but not *setTimeout( )*.
Object constructor	In Flash 5 ActionScript, the constructor for the *Object* class does not accept any parameters. In ECMA-262 and Flash 6 ActionScript, *Object* accepts a `value` parameter, which can be a Boolean, string, or number primitive.
Slash and Tell Target notation	ActionScript supports legacy slash notation for composing variable references and controlling movie clips via *Tell Target* (e.g., "/ball:speed "). Neither *Tell Target* nor slash notation are defined by ECMA-262.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Nonstandard event handlers *on( )* and *onClipEvent( )*	ActionScript supports a non-EMCA-262 event handler format for buttons (*on( )*) and movie clips (*onClipEvent( )*), used to attach event handlers directly to objects at authoring time. See Chapter 10.
`arguments.caller`	The `arguments.caller` property is supported by ActionScript, but it was deprecated from JavaScript in version 1.3. In Netscape 4, JavaScript's `arguments.caller` property refers to the *Arguments* object of the calling function. In ActionScript, the `arguments.caller` property is a reference to the calling function itself, not the caller's *Arguments* object. See `arguments.caller` in the *Language Reference* for details.
Undeclared variables	In JavaScript, a reference to an undeclared variable causes a runtime error. ActionScript leniently permits undeclared variable references.
`_ _proto_ _` property	ActionScript formally documents the *`Object`*`._ _proto_ _` property, which is not exposed by ECMA or JavaScript. See *`Object`*`._ _proto_ _` in the ActionScript Language Reference.
Invoking undeclared functions	In JavaScript, invoking an undeclared function causes a runtime error. In ActionScript, it fails silently.
`constructor` property	In JavaScript (in accordance with ECMA-262), the `constructor` property is always read from an object's constructor's `prototype` property (*`someObject`*`._ _proto_ _. constructor`). In Flash Player 6 and Flash Player 5, the `constructor` property is written directly to *`someObject`*. For a complete discussion of this subtle deviation, see Section 12.7.3.
Object.toString( )	ECMA-262 states that, for built-in classes, *toString( )* should return: [object *`class`*], where *`class`* is the name of the object's class. In ActionScript, *toString( )* returns [object Object] unless the class implements custom *toString( )* behavior (e.g., the *Date* object returns a human-readable date).
super "operator"	ActionScript's *super* "operator" is not an implementation of ECMAScript v4's *super* operator. See Section 5.12.8.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Part III: Appendixes

Appendix E. HTML Support in Text Fields

This appendix discusses special usage notes for HTML-enabled text fields and lists the HTML tags they support.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Appendix E. HTML Support in Text Fields

E.1 HTML Tag Support in Text Fields

The set of HTML tags supported by Flash 5 text fields includes: `<B>` , `<I>` , `<U>` , `<FONT>` , `<P>` , `<BR>` , and `<A>` . Flash 6 adds support for `<LI>` and `<TEXTFORMAT>` . Although the `<TABLE>` tag is not supported, HTML tables can be simulated with text field tab stops (see *`TextFormat .tabStops`* in the ActionScript Language Reference) and Macromedia's FDataGrid Component available at the Flash Exchange (http://www.macromedia.com/exchange/flash/).

To set HTML-styled text for a *TextField* object, set its *`TextField .html`* property to `true` and then set its *`TextField .htmlText`* property to the desired HTML-tagged text. The `<HTML>` and `<BODY>` tags are neither supported nor required. Unsupported tags are ignored, although their text content is preserved. Flash requires quotes around the values assigned to tag attributes. For further details on these and related topics, see Section E.5, Section E.4, and Section E.3 later in this appendix.

For more details on using HTML in text fields, see *`TextField .htmlText`* in the ActionScript Language Reference.

E.1.1 <B> (Bold)

The `<B>` tag renders text in bold, provided that a boldface style exists for the font in question. For example:

```
<B>This is bold text.</B>
```

will be rendered as, **This is bold text** .

See the discussion of bold font embedding under *`TextField .embedFonts`* and *`TextField .htmlText`* in the ActionScript Language Reference.

E.1.2 <I> (Italics)

The `<I>` tag renders text in italics, provided that an italic style exists for the font in question. For example:

```
<I>This is italic text.</I>
```

will be rendered as, *This is italic text* .

See the discussion of italic font embedding under *`TextField .embedFonts`* and *`TextField .htmlText`* in the ActionScript Language Reference.

E.1.3 <U> (Underline)

The `<U>` tag renders the tagged text with an underline beneath it. For example:

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

```
<U>This is underlined text.</U>
```

will be rendered as, This is underlined text .

Because hyperlinks are not underlined automatically in Flash, you should use the <U> tag to identify hyperlinks. For example:

```
<A HREF="http://www.flight404.com"><U>Click here</U>
</A> to visit a neat site.
```

will be rendered as, "Click here to visit a neat site."

E.1.4 <LI> (List Item)

The <LI> tag creates a bullet in front of the text it encloses. Unlike its HTML counterpart, <LI> does not require a <UL> tag. For example:

```
While you work: <LI>Stretch</LI><LI>Drink water</LI><LI>Rest your eyes</LI>
```

will be rendered as:

While you work:

- Stretch
- Drink water
- Rest your eyes

The <LI> tag does not support multilevel bullets or the <OL> (numbered list) tag. Lines formatted with *TextFormat* .bullet are represented with <LI> tags in *TextField* .htmlText .

E.1.5 <FONT> (Font Control)

The <FONT> tag supports the following three attributes:

FACE

The FACE attribute specifies the name of the font to use. In Flash 6, a list of multiple font faces can also be used (as in HTML), though this feature is undocumented. Flash attempts to render the first available font listed in the FACE attribute. For example, in the code <FONT FACE="Arial,Helvetica">my text</FONT> , Flash will render "my text" in Helvetica if Arial is missing but Helvetica is present. See *TextFormat* .font for details. For a function that picks an appropriate font from the user's system, see *TextField.getFontList( )* .

SIZE

The SIZE attribute specifies the size of the tagged text as a fixed point size (such as <FONT SIZE="18">) or as a relative size. Relative point sizes are preceded by a + or - sign and are specified relative to the text size in the text field Property inspector. For example, if the point size is 14 in the Property inspector, then <FONT SIZE="-2"> displays the tagged text at 12-point. See also *TextFormat* .size .

COLOR

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

> The `COLOR` attribute specifies the color of the tagged text, as a hexadecimal number, preceded by the pound sign (#)—for example, `<FONT COLOR="#FF0000">this is red text</FONT>` . Specify the hexadecimal number as an RGB series of three two-digit numbers from `00` to `FF` . Note that Flash's implementation of the `COLOR` attribute is more strict than HTML allows—the pound sign (#) is required, and color names such as "`green` " and "`blue` " cannot be used as `COLOR` values. See also the *Color* class and *`TextFormat` `.color`* .

Here are some `<FONT>` examples:

```
<FONT FACE="Arial">this is Arial</FONT>
<FONT FACE="Arial" SIZE="12">this is 12pt Arial</FONT>
<FONT FACE="Lucida Console" SIZE="+4" COLOR="#FF0000">this is red,
+4pt Lucida Console</FONT>
```

E.1.6 <P> (Paragraph Break)

The `<P>` tag demarcates paragraphs, but in Flash it behaves quite differently than its HTML counterpart. First of all, unterminated `<P>` tags do not cause line breaks in Flash as they do in regular HTML. Note the difference between Flash and web browser output for the following HTML-tagged text:

```
I hate filling out forms. <P>So sometimes I don't.
```

Flash output:

> I hate filling out forms. So sometimes I don't.

Web browser output:

> I hate filling out forms.
> So sometimes I don't.

Closing `</P>` tags are required by Flash in order for line breaks to be added. For example:

```
<P> I hate filling out forms.</P> So sometimes I don't.
```

Furthermore, in Flash, `<P>` causes a single line break, exactly like `<BR>` , whereas in web browsers, `<P>` causes a double line break. Consider the following:

```
<P>This is line one.</P><P>This is line two.</P>
```

In Flash, this code is rendered with no gap between the lines, as in:

> This is line one.
> This is line two.

In a web browser, the code is rendered with a gap between the lines, as in:

> This is line one.
>
> This is line two.

Because Flash's `<P>` tag behavior differs from its behavior in web browsers, we often use the `<BR>` tag instead. However, the `ALIGN` attribute of the `<P>` tag is still useful to center, right-justify, or left-justify text, as follows:

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

```
<P ALIGN="CENTER">Centered text</P>
<P ALIGN="RIGHT">Right-justified text</P>
<P ALIGN="LEFT">Left-justified text</P>
```

E.1.7
 (Line Break)

The `<BR>` tag causes a line break in a body of text and is functionally equivalent to the `\n` escape sequence or the `newline` keyword. Consider the following:

```
This is line one. <BR>This is line two.
This is line one. \nThis is line two.
```

Both are rendered in Flash as:

> This is line one.
> This is line two.

E.1.8 <A> (Anchor or Hypertext Link)

The `<A>` tag creates a hypertext link. When the user clicks text tagged with `<A>` , the document specified in the `HREF` attribute of the tag loads into the browser. If the movie is running in the Standalone Player, the default web browser on the system is launched and the document is loaded into that browser.

The generic syntax of the `<A>` tag is:

```
<A HREF="documentToLoad.html">linked text</A>
```

For example, to link to a good game console, we can use:

```
<A HREF="http://www.gamecube.com/">nice system</A>
```

As with HTML, the URL can be absolute or relative to the current page. Because Flash doesn't automatically add underlining or color to hyperlinks, you might want to augment the link tag like this:

```
<FONT COLOR="#0000FF">
<U><A HREF="http://www.gamecube.com/">nice console</A></U></FONT>
```

Normally, links followed from an anchor tag cause the current movie to be replaced with the document specified in the `HREF` of the anchor tag. However, an anchor tag may also cause a secondary browser window to launch. Using the `TARGET` attribute, we can specify the name of a window into which to load the linked document, as follows:

```
<A HREF="documentName" TARGET="windowName">linked text</A>
```

If a window named *`windowName`* does not already exist, the browser launches a new window and assigns it the name *`windowName`* . To launch a new document in its own, anonymous window, we can use the "`_blank` " keyword, as in:

```
<A HREF="mypage.html" TARGET="_blank">linked text</A>
```

Note that when we launch windows through the `TARGET` attribute, we have no control over the size or toolbar arrangement of the new window. To launch specifically sized windows from a link, we must use JavaScript. Techniques for launching customized secondary windows with JavaScript are described at:

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

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

For more information on communicating with JavaScript from ActionScript, see the global functions *fscommand ( )* and *getURL ( )* in the ActionScript Language Reference and Section E.6 later in this appendix.

In addition to "`_blank` ", the TARGET attribute can also be "`_parent` ", "`_self` ", or "`_top` ", or the name of a window or frame. For example, the `TARGET` attribute can load a document into a frame using:

```
<A HREF="documentName" TARGET="frameName">linked text</A>
```

Flash anchor tags do not always behave exactly like HTML anchor tags. For example, we cannot use the `NAME` attribute of the anchor tag in Flash, so internal links within a body of text are not possible.

In Flash, anchor tags are not added to the tab order of the movie and are therefore not accessible via the keyboard. If your content must be accessible to keyboards and alternative input devices, you should use buttons, not anchor tags, for links.

E.1.9 <TEXTFORMAT> (Formatting Support)

The `<TEXTFORMAT>` tag is a special Flash-only HTML extension used to implement the *TextFormat* class's paragraph formatting features in HTML syntax. Its attributes correspond to the *TextFormat* class's `leftMargin` , `rightMargin` , `blockIndent` , `indent` , `leading` , and `tabStops` properties, as follows:

LEFTMARGIN

Sets the left margin, in pixels, of the contained text. See *`TextFormat .leftMargin`* .

RIGHTMARGIN

Sets the right margin, in pixels, of the contained text. See *`TextFormat .rightMargin`*

BLOCKINDENT

Sets the left indentation, in pixels, of the contained text. See *`TextFormat .blockIndent`* .

INDENT

Sets the indentation, in pixels, for the first line of each contained paragraph. See *`TextFormat .indent`* .

LEADING

Sets the space, in pixels, between lines of the contained text. See *`TextFormat .leading`* .

TABSTOPS

Sets the horizontal tab stops for the contained text, as a comma-separated list of nonnegative integers. See *`TextFormat .tabStops`* .

The following code shows how to create a text field whose lines of text are separated vertically by 10 pixels and whose paragraphs' first lines are indented by 20 pixels:

```
this.createTextField("output_txt", 1, 0, 0, 100, 200);
output_txt.html = true;
output_txt.border = true;
output_txt.wordWrap = true;
output_txt.type = "input";
output_txt.htmlText = "<TEXTFORMAT LEADING='10' INDENT='20'>"
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

```
+ "<P>This is the first paragraph.</P>"
+ "<P>This is the second paragraph.</P>"
+ "<P>And finally, this is the third paragraph.</P>"
+ "</TEXTFORMAT>";
```

E.1.10 <TAB> (Tab Character)

The <TAB> tag (undocumented) is equivalent to inserting a tab character (ASCII 9) in a string. It is used to advance to the next tab stop when *TextField* .condenseWhite is true for the text field (in which case, literal tab characters are condensed with the rest of the whitespace).

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Appendix E. HTML Support in Text Fields

E.2 Entity Support

Flash HTML text fields also support the special character entities listed in Table E-1. Wherever an entity appears in a text field's *TextField*.htmlText property, Flash displays the corresponding character on screen. However, other common entities, such as ©, and numeric entities, such as ™ (trademark symbol), are not supported. To include characters such as the copyright symbol, the trademark symbol, or the euro, use Unicode-format text files, a Unicode escape sequence, or the *String.fromCharCode( )* method (see Section 4.5.1). The following example shows how to include a euro symbol in an HTML text field:

```
this.createTextField("price_txt", 0, 100, 100, 200, 30);
price_txt.html = true;
// The euro is represented by the escape sequence \u20AC
price_txt.htmlText = "<B>FINAL COST:</B> \u20AC 200";
```

Table E-1. Supported entities

Entity	Character Represented
<	<
>	>
&	&
"	"
'	'
	nonbreaking space

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Appendix E. HTML Support in Text Fields

E.3 Quoting Attribute Values

Outside of Flash, HTML attribute values can be quoted with single quotes, double quotes, or no quotes at all. The following tags are all valid in most web browsers:

```
<P ALIGN=RIGHT>
<P ALIGN='RIGHT'>
<P ALIGN="RIGHT">
```

But in Flash, unquoted attribute values are not allowed. For example, the syntax `<P ALIGN=RIGHT>` is illegal in Flash. However, both single and double quotes can be used to delimit attribute values. When composing text field values that include HTML attributes, we must be careful to quote attributes correctly, using one type of quote to demarcate the string itself and another to demarcate attribute values. For example:

```
// These examples are both valid.
myText = "<P ALIGN='RIGHT'>hi there</P>";
myText = '<P ALIGN="RIGHT">hi there</P>';
// This example causes an error because double quotation marks are
// used to demarcate both the string and the attribute.
myText = "<P ALIGN="RIGHT">hi there</P>";
```

For more information on using quotation marks to form strings, see Section 4.5.2.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Appendix E. HTML Support in Text Fields

E.4 Unrecognized Tags and Attributes

Like web browsers, Flash ignores tags and attributes it does not recognize. For example, if we assign the following value to an HTML text field in Flash:

```
<P>Please fill in and print this form</P>
<FORM><INPUT TYPE="TEXT"></FORM>
<P>Thank you!</P>
```

the output will be:

> Please fill in and print this form
> Thank you!

The `FORM` and `INPUT` elements are not supported by Flash, so both are ignored. Similarly, if we use container elements such as `<TD>`, the content is preserved but the markup is ignored. For example:

```
myTextField = "<TABLE><TR><TD>table cell text</TD></TR></TABLE>";
```

outputs the following line without table formatting:

> table cell text

However, in Flash 5 and Flash 6, if a tag is not terminated, the entire text that follows is considered part of the tag and will not display on screen. For example, given the following assignment:

```
theField_txt.htmlText = "We all know that 5 < 20. That's obvious.";
```

Flash displays:

> We all know that 5

To include a `<` character in an HTML text field, use the entity `<` as follows:

```
theField_txt.htmlText = "We all know that 5 &lt; 20. That's obvious.";
```

Flash displays:

> We all know that 5 < 20. That's obvious.

For more information on including HTML source code in an HTML text field, see:

> http://moock.org/asdg/technotes/sourceInHtmlField/

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Appendix E. HTML Support in Text Fields

E.5 Using HTML for Input and Output

HTML text entered manually into a text field using the Text tool in the Flash authoring tool will not be rendered as HTMI display HTML-formatted text on screen, we must assign HTML text to a *TextField* object's `htmlText` property or, in F the variable associated with the text field. For example:

```
// Flash 6 syntax
theField_txt.htmlText = "<P><B>Error!</B> <I>No email address!</I></P>";

// Flash 5 syntax
theField_txt = "<P><B>Error!</B> <I>No email address!</I></P>";
```

> In Flash 6, the Flash 5 syntax will not only fail to work, it will replace the *TextField* object with the assigned string! In Flash 6, use of the `htmlText` property is mandatory.

In the Flash Player, HTML can also be entered into a movie via an HTML-enabled or regular (non-HTML) user-input tex

When regular text is entered into an HTML-enabled user-input text field, the field's `htmlText` property includes marku that are added automatically. For example, the input text "Hi there" is converted to the `htmlText` property value:

```
'<P ALIGN="LEFT"><FONT FACE="Arial" SIZE="10" COLOR="#000000">Hi there</FONT
```

When the HTML characters <, >, `&` , ", and ' are entered into an HTML-enabled user-input text field, the field's `htmlTe` property represents them with the entities `>` , `<` , `&` , `"` , and `'` . For example, the input tex "`<B>hi there</B>` " is converted to the `htmlText` property value:

```
'<P ALIGN="LEFT"><FONT FACE="Arial" SIZE="10"
 COLOR="#000000">&lt;B&gt;hi there&lt;/B&gt;</FONT></P>'
```

In both cases, the original (non-marked-up) value is available via the `text` property.

When regular or HTML text is entered into a normal (non-HTML) user-input text field, no modification of the entered tex Regular user-input text fields allow raw HTML code to be entered into a movie without distortion. This is useful for show receiving HTML tags verbatim when you don't want them ignored, rendered, or interpreted.

An example showing HTML-enabled and regular user-input text field data entry is available at the online Code Depot.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Appendix E. HTML Support in Text Fields

E.6 Executing JavaScript from HTML Links

In most JavaScript-capable web browsers, it is possible to execute JavaScript statements from an anchor tag using the `javascript` : protocol as the value of the `HREF` attribute. For example:

```
<A HREF="javascript:square(5);">find the square of 5</A>
```

In ActionScript, we can also execute JavaScript statements from an `<A>` tag, like this:

```
theField_txt.htmlText = "<A HREF='javascript:alert(5);'>display the number 5
```

However, to include string values in JavaScript statements, we must use the HTML entity `"` for quotation marks, a

```
theField_txt.htmlText = "<A HREF=
                        'javascript:alert(&quot;hello world&quot;);'>"
                        + "display hello world</A>";
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Appendix E. HTML Support in Text Fields

E.7 Calling ActionScript Functions from HTML Links

Though arbitrary statements of ActionScript code cannot be executed from a Flash `<A>` tag, ActionScript *functions* can. To invoke an ActionScript function from an anchor tag, we use the following syntax:

```
<A HREF="asfunction:myFunctionName">invoke the function</A>
```

The function invocation operator `( )` is not allowed and should not be used when invoking an ActionScript function from an anchor tag. The function name must immediately follow the keyword `asfunction:`—if a space character occurs before the function name, the function will not execute.

In addition to calling an ActionScript function from an anchor tag, we can also pass one parameter to that function using the syntax:

```
<A HREF="asfunction:myFunctionName,myParameter">invoke the function</A>
```

where *`myParameter`* is the value of the parameter to pass. Inside the invoked function, *`myParameter`* is always a string. To pass more than one piece of information to a function from an anchor, we use a delimiter in the *`myParameter`* value and dissect the string ourselves in the function. For example, here's a function invocation that passes two values, separated by a | character, to the *roleCall( )* function:

```
<A HREF="asfunction:roleCall,megan|murray">invoke the function</A>
```

And here's the *roleCall( )* function. Notice how it separates the values with the *split( )* method:

```
function roleCall (name) {
  var bothNames = name.split("|");
  trace("first name: " + bothNames[0]);  // Displays: megan
  trace("last name: " + bothNames[1]);   // Displays: murray
}
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Part III: Appendixes

Appendix F. Support for GET and POST

Table F-1 lists the twelve functions and methods that can transfer content to an HTTP server in an HTTP request using either the GET or POST method. Due to limitations and bugs discussed in this appendix, for best results, you should use the HTTP GET method whenever possible, particularly when developing Flash 5-format movies. For Macromedia's information on POST-specific problems in the Flash Player, see:

http://www.macromedia.com/support/flash/ts/documents/flash_get_post.htm
http://www.macromedia.com/support/flash/ts/documents/flashplayer_r40.htm

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Appendix F. Support for GET and POST

F.1 Browser-Based Support for GET and POST

On both Windows and Macintosh, the browser plugin and ActiveX control versions of Flash Player 6 fully support the HTTP GET and POST methods. However, despite this support, minor bugs in certain browser/Player combinations cause intermittent failure for functions using POST (see preceding URLs). Furthermore, prior to version 6.0.40.0, the Flash Player's POST method caused problems for Java servlets, as discussed at:

http://www.macromedia.com/support/flash/ts/documents/java_get.htm

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Appendix F. Support for GET and POST

F.2 Standalone Player Support for GET and POST

Table F-1 lists, by function, the official support for HTTP GET and POST on Windows and Macintosh in Standalone Player 5 and Standalone Player 6. Note the following general rules:

- Standalone Player 5 on Windows and Macintosh does not support POST.
- Standalone Player 6 on Windows does not support POST for functions that receive results in a web browser rather than the Player.
- Hence, even if POST works on your system, you should use GET wherever possible for the best results on the most platforms.

Table F-1. Support for GET and POST as HTTP methods in Flash Standalone Player 5 and Standalone Player 6 for Windows and Macintosh

Function	Win GET F5	Mac GET F5	Win POST F5	Mac POST F5	Win GET F6	Mac GET F6	Win POST F6	Mac POST F6
getURL( )				[1]			[2]	
loadMovie( )				[1]				
loadMovieNum( )				[1]				
loadVariables( )		[3]		[1] [3]				
loadVariablesNum( )		[3]		[1] [3]				
LoadVars.send( )	N/A	N/A	N/A	N/A			[2]	
LoadVars.sendAndLoad( )	N/A	N/A	N/A	N/A				
MovieClip.getURL( )				[1]				
MovieClip.loadMovie( )				[1]				
MovieClip.loadVariables( )				[1]				
XML.send( )				[1]			[2]	
XML.sendAndLoad( )				[1]				

[1] POST requests are converted to GET requests but no variables are sent (i.e., the query string is missing). In

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

my tests, POST works in Standalone Player 5 on Macintosh, but the consensus from Macromedia and the developer community is that POST does not work reliably in that Player.

[2] POST requests are converted to GET requests, and data is transferred via the GET request query string.

[3] The entire *loadVariables( )* method works neither in the Standalone Player nor with executable Projectors (i.e., movies bundled with the Standalone Player in a single executable file).

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Part III: Appendixes

Appendix G. Flash UI Component Summary

This appendix lists and briefly describes the methods and properties supported by the Flash User Interface (FUI) Components that are installed with Flash MX. The FUI Components provide typical user interface elements, such as radio buttons, checkboxes, and lists. We chose to separate this summary of methods and properties in this appendix for several reasons:

- The ActionScript Language Reference is designed for full, detailed coverage of all the methods and properties of a class. Here we only summarize the methods and properties rather than documenting them fully. Full and proper coverage deserves more space (perhaps hundreds of pages) than we can offer within the constraints of this book.

- This book documents the core ActionScript language. The FUI Components are actually extensions to Flash and ActionScript and are themselves written in ActionScript. They are important examples of an infinite number of extensions that cannot possibly be covered in one book. For example, see Macromedia's *FDataGrid* Component available at the Flash Exchange (http://www.macromedia.com/exchange/flash/).

- This appendix serves as a quick reference for all the FUI Component methods and properties; you can leave it open while you work so that you won't have to check Flash's online Help or pulldown menus to see if a method or property exists. We've also organized the components differently (such as in Table G-1 and Table G-2) and provided more complete synopses than Macromedia's online Help provides.

- We felt it was important to group all the FUI Components together conceptually, rather than alphabetically in the *Language Reference* where they would be interrupted by other items such as the `_focusrect` global property and *fscommand( )* function.

For complete Flash UI Component documentation, consult the resources listed under Section 14.7 at the end of Chapter 14.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Appendix G. Flash UI Component Summary

FCheckBox — support for a standard UI checkbox element

Methods

getEnabled( )

Boolean indicating whether a checkbox is enabled (i.e., responds to mouse input).

getLabel( )

Returns a checkbox's string label.

getValue( )

Boolean indicating whether a checkbox is checked (i.e., selected or deselected).

registerSkinElement( )

Registers a skin element for changing the appearance of a checkbox.

setChangeHandler( )

Specifies the callback function to call when the checkbox's state changes.

setEnabled( )

Enables or disables the checkbox (not the same as selecting or deselecting it).

setLabel( )

Sets a checkbox's string label.

setLabelPlacement( )

Specifies the label location relative to the checkbox ("left" or "right").

setSize( )

Sets the width of the checkbox element, in pixels.

setStyleProperty( )

Sets a particular style property for the check box (for a list of supported properties, see the *FStyleFormat* object).

setValue( )

Specifies whether the checkbox is checked (`true`) or unchecked (`false`) (i.e., whether it is selected or deselected).

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Appendix G. Flash UI Component Summary

FComboBox and FListBox — support for standard UI combo boxes (i.e., drop-down lists) and list boxes

---

Table G-1 juxtaposes the methods of the *FComboBox* and *FListBox* objects. Note how they share the majority of methods, although *FListBox* supports selecting multiple line items simultaneously

Table G-1. FComboBox and FListBox methods

Method	Description	FComboBox	FListBox
addItem( )	Appends an item to the list of items in the box		
addItemAt( )	Adds an item at the specified position in the list		
getEnabled( )	Boolean indicating whether the element is enabled or dimmed ("grayed-out")		
getItemAt( )	Returns the list item at the specified position		
getLength( )	Returns the number of items in the list		
getRowCount( )	Returns the number of rows displayed in a scrolling list		
getScrollPosition( )	Returns the line number of the top row displayed		
getSelectedIndex( )	Returns the index of the current list selection		
getSelectedIndices( )	Returns an array of indexes of, potentially, multiple selections		
getSelectedItem( )	Returns the selected list item as an object		
getSelectedItems( )	Returns an array of selected items as objects		
getSelectMultiple( )	Boolean indicating whether multiple items can be selected in a list box		
getValue( )	Returns the value of the selected item		
registerSkinElement( )	Registers a skin element for changing the component appearance		
removeAll( )	Removes all items from the list		
removeItemAt( )	Removes a specified item in the list		

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

replaceItemAt( )	Replaces the item at the specified position in the list		
setAutoHideScrollBar( )	Boolean specifying whether to hide the scroll bar when it isn't needed		
setChangeHandler( )	Specifies the callback function to call when the list's state changes		
setDataProvider( )	Specifies an array of elements or a *DataProvider* object to populate the list		
setEditable( )	Boolean specifying whether the combo box's contents are editable		
setEnabled( )	Enables or disables (i.e., deactivates) the list box		
setItemSymbol( )	Specifies the Library symbol ID of a graphics symbol for displaying list content		
setRowCount( )	Specifies the maximum number of rows displayed in a list		
setScrollPosition( )	Causes the list box to scroll to the specified item		
setSelectedIndex( )	Selects an item in the list by its row index (zero-relative)		
setSelectedIndices( )	Specifies an array of items (by index) to be selected		
setSelectMultiple( )	Boolean specifying whether multiple items can be selected in a list box		
setSize( )	Sets the width of the combo box or list box, in pixels (can also set height of a list box)		
setStyleProperty( )	Sets a particular style property for the combo box or list box (for a list of supported properties, see the *FStyleFormat* object)		
setValue( )	Sets the current value for a combo box		
setWidth( )	Sets the list box width, in pixels		
sortItemsBy( )	Sorts the items in the list by either their labels or the underlying data		

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Appendix G. Flash UI Component Summary

FPushButton — support for a standard UI pushbutton element

Methods

getEnabled( )

Boolean indicating whether a pushbutton is enabled (i.e., responds to mouse input).

getLabel( )

Returns a pushbutton's string label.

registerSkinElement( )

Registers a skin element for changing the appearance of a pushbutton.

setClickHandler( )

Specifies the callback function to call when the pushbutton is clicked.

setEnabled( )

Enables or disables the pushbutton (i.e., specifies whether it is active or dimmed).

setLabel( )

Sets a pushbutton's string label.

setSize( )

Sets the height and width of the pushbutton, in pixels.

setStyleProperty( )

Sets a particular style property for the pushbutton (for a list of supported properties, see the *FStyleFormat* object).

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Appendix G. Flash UI Component Summary

FRadioButton and FRadioButtonGroup	support for standard UI radio buttons and groups

Table G-2 juxtaposes the methods of the *FRadioButton* and *FRadioButtonGroup* objects. Note how they share several methods, although in the case of *FRadioButtonGroup* objects, the method pertains to the radio button group as a whole and not an individual button.

Table G-2. FRadioButton and FRadioButtonGroup methods

Method	Description	FRadioButton	FRadioButtonGroup
getData( )	Returns the data associated with a radio button		
getEnabled( )	Boolean indicating whether a radio button or group is enabled (i.e., is active or dimmed)		
getGroupName( )	Returns the radio button group name		
getLabel( )	Returns the label of a radio button		
getState( )	Returns the state of a radio button—selected (`true`) or not selected (`false`)		
getValue( )	Returns the value of the selected button in a group		
registerSkinElement( )	Registers a skin element for changing the appearance of a radio button or group		
setChangeHandler( )	Specifies the callback function to call when the radio button or group changes state		
setData( )	Associates a datum with a radio button		
setEnabled( )	Enables or disables a radio button or group		
setGroupName( )	Sets the name of a radio button group		
setLabel( )	Sets the string label of a radio button		
setLabelPlacement( )	Specifies the label location for a radio button or group ("left" or "right")		
setSize( )	Sets the width of a radio button or group, in pixels		

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

setState( )	Boolean specifying the radio button state—selected (`true`) or not selected (`false`)		
setStyleProperty( )	Sets a particular style property for the radio button or group (for a list of supported properties, see the *FStyleFormat* object)		
setValue( )	Selects a radio button within a group based on its data value		

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Appendix G. Flash UI Component Summary

FScrollBar	support for a standard UI scrollbar element associated with a text field using setScrollTarget( )

Methods

getEnabled( )

Boolean indicating whether a scrollbar is enabled (i.e., responds to mouse input).

getScrollPosition( )

Returns the current scroll position as an integer between the minimum and maximum positions.

registerSkinElement( )

Registers a skin element for changing the appearance of the scrollbar.

setChangeHandler( )

Specifies the callback function to call when the scrollbar is scrolled.

setEnabled( )

Enables or disables a scrollbar.

setHorizontal( )

Boolean specifying whether the scrollbar should be horizontal (`true`) or vertical (`false`).

setLargeScroll( )

Specifies the number of units to scroll when the user clicks to either side of the thumb slider (defaults to `pageSize` as set with *setScrollProperties( )*).

setScrollPosition( )

Sets the current scroll position as an integer between the minimum and maximum positions.

setScrollProperties( )

Sets the scrollbar's minimum and maximum positions and page size.

setScrollTarget( )

Specifies the text field that the scrollbar causes to scroll (i.e., the text field to which the scrollbar applies), as a string instance name.

setSize( )

Sets the scrollbar length in pixels (its width is fixed).

setSmallScroll( )

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Specifies the number of units to scroll when the user clicks a scroll arrow (defaults to 1).

setStyleProperty( )

Sets a particular style property for the scrollbar (for a list of supported properties, see the *FStyleFormat* object).

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Appendix G. Flash UI Component Summary

FScrollPane — support for creating a scrolling view of a movie, movie clip, or JPEG image

Methods

getPaneHeight( )

Returns the height of a scroll pane as set by *setSize( )*.

getPaneWidth( )

Returns the width of a scroll pane as set by *setSize( )*.

getScrollContent( )

Returns a reference to the movie clip displayed in the scroll area.

getScrollPosition( )

Returns the current scroll position as an object with `x` an `y` properties.

loadScrollContent( )

Loads a *.swf* or *.jpg* file into a scroll pane for display.

refreshPane( )

Refreshes the scroll pane (useful if contents change in size).

registerSkinElement( )

Registers a skin element for changing the appearance of a scroll pane.

setDragContent( )

Boolean specifying whether the user can scroll the content in the pane by dragging.

setHScroll( )

Specifies whether a horizontal scroll bar is displayed always (`true`), never (`false`), or as needed (`"auto"`).

setScrollContent( )

Specifies a Library symbol or move clip instance to display in the scroll pane.

setScrollPosition( )

Sets the current scroll position using *x* and *y* coordinates.

setSize( )

Sets the scroll pane's width and height.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

setStyleProperty( )

Sets a particular style property for the scroll pane (for a list of supported properties, see the *FStyleFormat* object).

setVScroll( )

Specifies whether a vertical scroll bar is displayed always (`true`), never (`false`), or as needed (`"auto"`).

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Appendix G. Flash UI Component Summary

FStyleFormat support for applying styles to Flash UI Component elements

Methods

addListener( )

Adds UI Components to a list of items to be updated when the *FStyleFormat* changes.

applyChanges( )

Applies style changes and notifies (refreshes) all listeners.

removeListener( )

Removes a style format listener.

Properties

arrow

Sets the RGB color of arrows used in scroll bars, list boxes, etc.

background

Sets the RGB color of the background of associated UI components.

backgroundDisabled

Sets the background RGB color for disabled UI components.

check

Sets the RGB color for checkmarks in checkboxes.

darkshadow

Sets the dark shadow RGB color used in associated radio buttons, checkboxes, etc.

embedFonts

Specifies whether to embed fonts for text within UI components.

face

Sets the primary RGB color used for associated UI components.

focusRectInner

Sets the RGB color of the inner focus rectangle for UI components.

focusRectOuter

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Sets the RGB color of the outer focus rectangle for UI components.

foregroundDisabled

Sets the RGB color for the foreground of disabled UI components.

highlight

Sets the RGB color for highlights on UI components.

highlight3D

Sets the RGB color for 3D highlights on UI components.

radioDot

Sets the RGB color for the dots in radio buttons.

scrollTrack

Sets the RGB color for the background track of a scrollbar.

selection

Sets the RGB color for selections.

selectionDisabled

Sets the RGB color for disabled selections.

selectionUnfocused

Sets the RGB color for unfocused selections.

shadow

Sets the RGB color for shadows.

textAlign

Sets the text alignment ("right", "left", or "center").

textBold

Boolean specifying whether text should be bolded.

textColor

Sets the RGB color of text in the UI component.

textDisabled

Sets the RGB color for disabled text.

textFont

Sets the text font for UI components.

textIndent

Sets the integer indentation, in pixels, for text in UI components.

textItalic

Boolean specifying whether text should be italicized.

textLeftMargin

Sets the left margin, in pixels, for text in UI components.

textRightMargin

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Sets the right margin, in pixels, for text in UI components.

textSelected

Sets the RGB color of selected text in UI components.

textSize

Sets the font text size, in points.

textUnderline

Boolean specifying whether text should be underlined.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Part III: Appendixes

Appendix H. Embedding a Flash Movie in a Web Page

To embed a Flash movie in a web page we use the HTML <EMBED> and <OBJECT> tags, much like we use the <IMG Internet Explorer (IE) for Windows (IE5.5 SP2 and later) require the <OBJECT> tag (the <OBJECT> tag is also prefer Windows, which supported both tags). Internet Explorer for Macintosh and Netscape on both platforms require the <EMB browsers, we nest the <EMBED> tag within the <OBJECT> tag as shown in Example H-1, which embeds a movie called

Example H-1. <OBJECT> AND <EMBED> tags for embedding a .swf file in an HTML page

```
<OBJECT
  CLASSID="clsid:D27CDB6E-AE6D-11cf-96B8-444553540000"
  CODEBASE=
"http://download.macromedia.com/pub/shockwave/cabs/flash/swflash.cab#version
  WIDTH="550"
  HEIGHT="400"
  ID="main"
  ALIGN="">
  <PARAM NAME="movie" VALUE="main.swf">
  <PARAM NAME="quality" VALUE="high">
  <PARAM NAME="bgcolor" VALUE="#FFFFFF">

    <EMBED
      SRC="main.swf"
      QUALITY="high"
      BGCOLOR="#FFFFFF"
      WIDTH="550"
      HEIGHT="400"
      NAME="main"
      ALIGN=""
      TYPE="application/x-shockwave-flash"
      PLUGINSPAGE="http://www.macromedia.com/go/getflashplayer">
    </EMBED>
</OBJECT>
```

In the Flash authoring tool, you can use the File Publish command to generate <EMBED> and <OBJECT> tags autc Publishing Publishing Flash documents.

For more information on the <EMBED> and <OBJECT> tags, see Help Using Flash Publishing Editing F

http://www.macromedia.com/support/flash/ts/documents/tn4150.html
http://www.macromedia.com/support/flash/ts/documents/tag_attributes.htm

Despite the assistance offered by the Flash authoring tool, this appendix is for developers who prefer to hand-code their H

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

the authoring tool.

Table H-1 lists the minimum recommended attributes of the `<EMBED>` and `<OBJECT>` tags, some of which are strictly other attributes.

Table H-1. Recommended attributes of the <OBJECT> and <E

Description	<OBJECT> attribute	
Identifies content as Flash	CLASSID="clsid:D27CDB6E-AE6D-11cf-96B8-444553540000"	T
Location of the Flash Player browser plugin	CODEBASE="http://download.macromedia.com/pub/shockwave/cabs/flash/swflash.cab#version=6,0,0,0"	F
URL of *.swf* file	<PARAM NAME="MOVIE" VALUE="*url* ">	S
Height of the movie	HEIGHT="*height* "	H
Width of the movie	WIDTH="*width* "	W

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Appendix H. Embedding a Flash Movie in a Web Page

H.1 Attributes of the <OBJECT> Tag

To use attributes other than `CLASSID` , `CODEBASE` , `WIDTH` , `HEIGHT` , `ID` , and `ALIGN` with the `<OBJECT>` tag, `<PARAM>` tag of the following format:

```
<PARAM NAME=ATTR_NAME VALUE=VAL>
```

where *`ATTR_NAME`* is the name of the attribute and *`VAL`* is the attribute value (as shown in Example H-1).

The following list of attributes applies exclusively to the `<OBJECT>` tag:

CLASSID

Identifier for the Flash ActiveX control, which must be `"clsid:D27CDB6E-AE6D-11cf-96B8-444553` regardless of the version of the movie or Player.

CODEBASE

Location of the Flash ActiveX control, which must be `"http://download.macromedia.com/pub/shockwave/cabs/flash/swflash.cab#ver` . It is used by IE for Windows to download the ActiveX version of the Flash Player if it is not installed or if its vers version specified by `#version=w,x,y,z` . If `#version` is omitted, the browser does not attempt to upgrade ActiveX Player on the user's system, but it will attempt to download the latest Player if no Player is found.

ID

An identification string that can be used by scripts to manipulate the Flash movie. Should match the `NAME` attribut `<EMBED>` tag. See *fscommand( )* in the *Language Reference* and http://www.moock.org/webdesign/flash/fscomm

MOVIE

The location of the movie (*.swf* file) as a relative or absolute URL, such as "`mymovie.swf` ". Should match the for the `<EMBED>` tag.

See Table H-1 and Section H.3 later in this appendix for more information.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Appendix H. Embedding a Flash Movie in a Web Page

H.2 Attributes of the <EMBED> Tag

The following list of attributes applies exclusively to the `<EMBED>` tag:

NAME

An identification string that can be used by scripts to manipulate the Flash movie. Should match the `ID` attribute used for the `<OBJECT>` tag. See *fscommand( )* in the *Language Reference* and http://www.moock.org/webdesign/flash/fscommand.

PLUGINSPAGE

Location of the Flash plugin on Macromedia's web site, used by Netscape to prompt the user to download the plugin if it is not already installed. Should be `"http://www.macromedia.com/go/getflashplayer"`.

SRC

The location of the movie (*.swf* file) as a relative or absolute URL, such as `"mymovie.swf"`. Should match the `MOVIE` attribute used for the `<OBJECT>` tag.

SWLIVECONNECT

When `true`, the browser loads Java when loading the Flash Player for the first time. Defaults to `false` (browser doesn't load Java if it isn't already running). For movies that are controlled by JavaScript or that use *fscommand( )*, you must set `SWLIVECONNECT` to `true`.

TYPE

The MIME content type for Flash. Required by Netscape if the web server does not specify the MIME type for *.swf* files. If your HTML page loads the Flash Player in IE for Windows but not in other browsers, be sure you've set the `TYPE` properly. Must be `"application/x-shockwave-flash"`.

See Table H-1 and Section H.3 for more information.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Appendix H. Embedding a Flash Movie in a Web Page

H.3 Common Attributes of the <OBJECT> and <EMBED> Tags

The following list of attributes applies to the both `<OBJECT>` and `<EMBED>` tags. See Example H-1 for sample usage.

ALIGN

Positions the movie in the browser window. When omitted, defaults to center, but there is no explicit "CENTER" option. Possible values are: `""` (no value, indicating centered), `"LEFT"`, `"RIGHT"`, `"TOP"`, and "`BOTTOM`".

BASE

The URL to use as the base of all relative URLs used in the movie (e.g., URLs used by *getURL( )*, *loadMovie( )*, *XML.send( )*, etc.).

BGCOLOR

The background color of the movie, in RGB hexadecimal triplet notation—*#RRGGBB*, where *RR*, *GG*, and *BB* are two-digit hex numbers representing Red, Green, and Blue. Overrides any author-time setting made via Modify Document.

HEIGHT

Height of the movie, either in pixels or as a percentage of the HTML page width in the browser.

LOOP

Causes the Flash animation to loop continuously (if `true`, the default) or stop (if `false`) when the playhead reaches the end of the movie.

MENU

Indicates whether the Flash Player's contextual menu contains a full list of items (`true`) or a minimal list of items (`false`). It defaults to `true` but is often set to `false` to prevent users from manipulating the movie in undesirable ways, such as stopping it, rewinding it, or zooming it. The contextual menu is accessed using right-click under Windows or Ctrl-click on the Macintosh.

PLAY

When `true` (the default), the movie plays immediately upon loading. When `false`, the movie does not play and must be started via ActionScript using the *play( )* or *gotoAndPlay( )* function.

QUALITY

Sets the antialiasing of text and graphics to control playback quality and performance. Values are: `"BEST"`, `"HIGH"`, `"AUTOHIGH"`, `"MEDIUM"`, `"AUTOLOW"`, or `"LOW"`.

SALIGN

Dictates the positioning of the movie, relative to the space available to the Flash Player. For possible values and usage, see Table 18-20 under `Stage.align` in the *Language Reference*.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

SCALE

Controls scaling. Values are: `"EXACTFIT"`, `"NOBORDER"`, `"NOSCALE"`, or `"SHOWALL"`. For usage, see `Stage.scaleMode` in the *Language Reference*.

WIDTH

Width of the movie, either in pixels or as a percentage of the HTML page width in the browser.

WMODE

Allows Flash movies to be windowless for more seamless integration with non-Flash elements using HTML layers. Supported values are: "`WINDOW`" (the default), "`OPAQUE`", or "`TRANSPARENT`". Flash Player 6.0.61.0 adds support for the `WMODE` attribute under Mac OS X and Netscape for Windows. Prior versions supported `WMODE` only when using Internet Explorer for Windows. For details, see the release notes for Flash Player 6.0.61.0 or Help Using Flash.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Appendix H. Embedding a Flash Movie in a Web Page

H.4 Runtime Access Affecting HTML Attributes

Table H-2 lists the HTML attributes that relate to, or can be affected by, ActionScript at runtime

Table H-2. The <EMBED> and <OBJECT> tag runtime-related attributes

Runtime access	<OBJECT> attribute	<EMBED> attribute
To load additional movies into the Player, use *loadMovie( )*.	<PARAM NAME="MOVIE" VALUE="*url*">	SRC="*url*"
Use _root.`_url` to determine the location from which the movie was loaded.	BASE="*url*"	BASE="*url*"
Draw a colored rectangle using a `depth` of -16384 using *MovieClip.beginFill( )*, `Stage.height`, and `Stage.width`.	<PARAM NAME="bgcolor" VALUE="#FFFFFF">	BGCOLOR= ="#FFFFFF"
Requires Flash Player 6. See Section 2.6 for details, including alternatives for supporting Flash Player 5.	<PARAM NAME="FLASHVARS" VALUE="AUTHOR=Moock">	FLASHVARS="AUTHOR=Moock"
Use *gotoAndPlay( )* to play the movie starting at frame 1.	<PARAM NAME="LOOP" VALUE="TRUE">	LOOP="TRUE"
`Stage.showMenu` (requires Flash Player 6) For Flash Player 5, use *fscommand("showmenu", "false")* and *fscommand("showmenu", "true")*.	<PARAM NAME="MENU" VALUE="TRUE">	MENU="TRUE"
Use *stop( )* to halt the movie at frame 1.	<PARAM NAME="PLAY" VALUE="TRUE">	PLAY="TRUE"
`_quality`	<PARAM NAME="QUALITY" VALUE="HIGH">	QUALITY="HIGH"
`Stage.align` (requires Flash Player 6)	<PARAM NAME="SALIGN" VALUE="LT">	SALIGN="LT"
`Stage.scaleMode` (requires Flash Player 6) For Flash Player 5, use *fscommand("allowscale")*.	<PARAM NAME="SCALE" VALUE="NOSCALE">	SCALE="NOSCALE"

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ActionScript for Flash MX: The Definitive Guide, 2nd Edition
By Colin Moock

Colophon

Our look is the result of reader comments, our own experimentation, and feedback from distribution channels. Distinctive covers complement our distinctive approach to technical topics, breathing personality and life into potentially dry subjects.

The animal on the cover of *ActionScript for Flash MX: The Definitive Guide*, Second Edition, is a siren, a particular type of salamander found in North American freshwater habitats. Salamanders are amphibians with tails; they diverged from other amphibian species (including frogs) early in amphibian evolution. All salamanders have smooth skin that is kept moist by secretions from numerous glands. When their environment becomes too dry, too hot, or too cold, the cold-blooded salamanders burrow into mud, and their body functions slow down. Sirens develop cocoons to shelter themselves until conditions improve.

Sirens are considered very primitive salamanders, as they are aquatic (live permanently in water), lack hind legs, have reduced front legs, gills, and other larval features into maturity, and their offspring are fertilized externally in the water. They are active at night, coming out and swimming much like eels do, using their dorsal fin in side-to-side motion to propel them forward. As they do so, they feed by gulping at insects and larvae. They expel plant matter through their gills.

Like all amphibians, sirens are an important part of a balanced ecosystem. They are small predators who in turn are preyed upon by fish and birds. Their glandular skin and fragile systems put sirens in the unfortunate position of being early pollution indicators in their freshwater habitats. Siren intermedia, an unusual siren subspecies that inhabits a large North American range and is known for its ability to produce vocal clicks and shrills, has nearly disappeared from Michigan, perhaps due to the presence of Rotenone, a chemical tool used to manage fisheries.

Brian Sawyer was the production editor and copyeditor for *ActionScript for Flash MX: The Definitive Guide*, Second Edition. Linley Dolby, Jeff Holcomb, and Claire Cloutier provided quality control. Genevieve D'Entremont, Judy Hoer, and Andrew Savikas provided production support. Julie Hawks wrote the index.

Ellie Volckhausen designed the cover of this book, based on a series design by Edie Freedman. The cover image is a 19th-century engraving from the Dover Pictorial Archive. Emma Colby produced the cover layout with QuarkXPress 4.1 using Adobe's ITC Garamond font.

David Futato designed the interior layout. This book was converted to FrameMaker 5.5.6 by Joe Wizda with a format conversion tool created by Erik Ray, Jason McIntosh, Neil Walls, and Mike Sierra that uses Perl and XML technologies. The text font is Linotype Birka; the heading font is Adobe Myriad Condensed; and the code font is LucasFont's TheSans Mono Condensed. The illustrations that appear in the book were produced by Robert Romano and Jessamyn Read using Macromedia FreeHand 9 and Adobe Photoshop 6. The tip and warning icons were drawn by Christopher Bing. This colophon was written by Sarah Jane Shangraw and Brian Sawyer.

The online edition of this book was created by the Safari production group (John Chodacki, Becki Maisch, and Madeleine Newell) using a set of Frame-to-XML conversion and cleanup tools written and maintained by Erik Ray, Benn Salter, John Chodacki, and Jeff Liggett.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir