



Understanding Flash MX 2004 ActionScript 2

Basic techniques for creatives

```
onClipEvent (load) {  
  a = 1;  
  timesmorz = 0.92;  
  maxang = 60;  
  n_osc = 1;  
}  
onClipEvent (enterFrame) {  
  setProperty (_root.pend, _rotation, (((Math.sin (n_osc*a))*Math.pow (timesmorz, a))))*maxang);  
  a = a+0.1;  
}  
onClipEvent (load) {  
  a = 1;  
  timesmorz = 0.92;  
  maxang = 60;  
  n_osc = 1;  
}  
onClipEvent (enterFrame) {  
  setProperty (_root.pend, _rotation, (((Math.sin (n_osc*a))*Math.pow (timesmorz, a))))*maxang);  
  a = a+0.1;  
}  
  
if (Key.isDown(Key.CONTROL)) { UserCounter++;  
  root.insertDuplicateMovieClip ("UserCounter", UserCounter);  
  root["UserCounter"]._x+=10;  
}  
  
onClipEvent (enterFrame) {  
  if (Key.isDown(Key.RIGHT)) {  
    this._x+=moveSpeed;  
  } else if (Key.isDown(Key.LEFT)) {  
    this._x-=moveSpeed;  
  }  
  if (Key.isDown(Key.DOWN)) {  
    this._y+=moveSpeed;  
  } else if (Key.isDown(Key.UP)) {  
    this._y-=moveSpeed;  
  }  
}
```

ALEX MICHAEL



This Page Intentionally Left Blank

Understanding Flash MX 2004 ActionScript 2

Basic Techniques for Creatives

Alex Michael



ELSEVIER

AMSTERDAM • BOSTON • HEIDELBERG • LONDON • NEW YORK • OXFORD
PARIS • SAN DIEGO • SAN FRANCISCO • SINGAPORE • SYDNEY • TOKYO

Focal Press is an imprint of Elsevier



Focal Press
An imprint of Elsevier
Linacre House, Jordan Hill, Oxford OX2 8DP
200 Wheeler Road, Burlington MA 01803

First published 2004

Copyright © 2004, Alex Michael. All rights reserved

The right of Alex Michael to be identified as the author of this work
has been asserted in accordance with the Copyright, Designs and
Patents Act 1988

No part of this publication may be reproduced in any material form (including
photocopying or storing in any medium by electronic means and whether
or not transiently or incidentally to some other use of this publication) without
the written permission of the copyright holder except in accordance with the
provisions of the Copyright, Designs and Patents Act 1988 or under the terms of
a licence issued by the Copyright Licensing Agency Ltd, 90 Tottenham Court Road,
London, England W1T 4LP. Applications for the copyright holder's written
permission to reproduce any part of this publication should be addressed
to the publisher

Permissions may be sought directly from Elsevier's Science and Technology
Rights Department in Oxford, UK: phone: (+44) (0) 1865 843830;
fax: (+44) (0) 1865 853333; e-mail: permissions@elsevier.co.uk. You may
also complete your request on-line via the Elsevier homepage
(www.elsevier.com), by selecting 'Customer Support' and then 'Obtaining
Permissions'

British Library Cataloguing in Publication Data

A catalogue record for this book is available from the British Library

Library of Congress Cataloguing in Publication Data

A catalogue record for this book is available from the Library of Congress

ISBN 0 240 51931 0

For information on all Focal Press publications visit our website at: www.focalpress.com

Typeset by Keyword Typesetting Services
Printed and bound in Great Britain

This Page Intentionally Left Blank

Contents

<i>Acknowledgements</i>		vii
<i>Introduction – Understanding ActionScript: Basic Techniques for Creatives</i>		ix
Section 1	ActionScript Basics	1
Chapter 1	An Introduction to ActionScript Programming Tools in Flash MX 2004 and Professional	3
Chapter 2	Constructing and Sharing ActionScript	20
Chapter 3	Learning about Events and Handlers	35
Section 2	ActionScript Fundamentals	51
Chapter 4	Variables, “If”, “For” and “While” Loops	53
Chapter 5	Functions and Arrays	72
Section 3	Object Fundamentals	89
Chapter 6	Keyboard Object	91
Chapter 7	Sound Object	109
Chapter 8	Color Object	125
Chapter 9	XML Object	146
Section 4	Working with Components	163
Chapter 10	UI Components	165
Chapter 11	Trivia Quiz Game Built Using Components	187

Section 5	Game Building	201
Chapter 12	Goose Game	203
Chapter 13	ActionScript-driven Effects and Techniques	223
Appendices		245
Appendix A	ActionScript Terminology	247
Appendix B	Letters A to Z and Standard Numbers 0 to 9	250
<i>Index</i>		253

Acknowledgements

I would like to thank my wife Liz and my two daughters Ellen and Alex for all their support.

I would like to thank Erika Pelser for a great front cover design and the rest of the team at Sprite Interactive including Rob Vacher, Kay Siegert and Ben Salter.

I would also like to thank the team at Focal Press who made this book possible.

This Page Intentionally Left Blank

Introduction – Understanding ActionScript: Basic Techniques for Creatives

Since the release of Flash MX, scripting in Flash has moved from being a desirable asset to an essential skill in the world of web design. It has also become a whole lot more difficult, with all major advances with Flash MX being code based. In fact, many of MX's most powerful features are accessible only via ActionScript. If you've never coded in Flash, you're going to want to very soon. This book is designed for people new to ActionScript or people who have learned ActionScript in an ad hoc manner. ActionScript, although a scripting language, has the basic fundamentals of many programming languages. The following chapters take you through a step-by-step approach to scripting techniques, illustrated with highly visual examples throughout the book. A support website at www.sprite.net/understanding provides all the content you need to try out the techniques shown in the book for yourself.

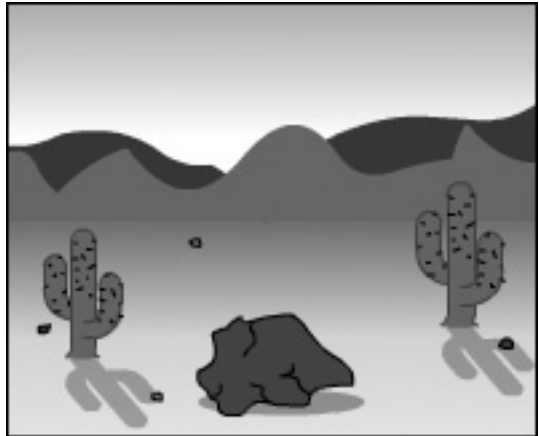
Ideal for those studying multimedia and information technology and anyone who wants to produce highly effective online interactive content, this book gives you all you need to ensure you have a firm foundation of knowledge on how to use ActionScript creatively so you can produce professional results. The practical tutorials demystify the use of ActionScript for effective Flash movie design, focusing on the designer artist.

ActionScript is a tool that worries most designers. This book will teach Flash designers how to create ActionScripts and will become a reference for all their design work. It will take you from knowing nothing about ActionScript to a level of understanding that will allow you to build complex interactive solutions, and build on good working practices that are reinforced by the developer community. Key features of the book include:

- It extends the basics step by step in a tutorial led approach.
- It builds good ActionScript programming foundations.
- It includes some very visual examples to show you what you can achieve.
- It comes with a free, fully commercial new game.
- It has an extensive support website – www.sprite.net/understanding

The book is split into five sections, with the first section focusing on an introduction to ActionScript, looking at the tools for creating and managing code. The second section covers the fundamentals of programming with the focus on the syntax of ActionScript. Section 3

introduces the key methods of ActionScript and how you string them together to build websites and applications. Sections 4 and 5 are made up of two walk-through examples – one on how to use components in a quiz game and the other is a shooting gallery style game. The final section also details how the various techniques explained in this book can be used to create a complex, commercial-quality game.



ACTIONSRIPT BASICS

01

This Page Intentionally Left Blank

1 An Introduction to ActionScript Programming Tools in Flash MX 2004 and Professional

Macromedia Flash MX 2004 or the Professional Version?

With the latest release of Macromedia Flash, you now have a choice between a standard version of Flash and a professional version. If you are currently a user of Flash MX then the 2004 upgrade will improve productivity, and by taking advantage of its extendable architecture will allow you to build complex applications. The professional version is built with specific features for developers. These features will be most noticeable if you are a developer working with Rich Internet Applications, deploy video to the web, or build content for mobile devices.

Flash MX Professional offers a new scripting window that helps you in organizing and writing external scripts. The script window allows you to create JavaScript files and Flash communications files. These files open like a text editor that supports code hints, syntax checking, formatting and code coloring. I like the document tabs that unfortunately are not available to Mac users. These are great for navigating between code and the Flash document.

Overall, Flash MX Professional offers features that designers and developers at any level can use. My personal view is that the professional version is probably easier for anybody new to programming as it offers you extra and more useful tools. If you are unsure which version to buy then download a free 30 day trial and see for yourself. To keep things simple I will in this book work on code that on the whole looks and is created in a similar way in both environments.

What is ActionScript?

ActionScript is a programming language, very similar to JavaScript. The purpose of ActionScript is to give instructions to your movie. These instructions are in the form of a script made up of a statement or a series of statements that execute specific tasks. Statements are lines of code that you attach to either a Button Instance or a keyframe. When the user clicks a button the statements attached to that button are executed. When the playhead passes over a particular keyframe with a script attached, the statements attached to the keyframe are executed. For example, the following is a statement attached to a keyframe:

```
stop ( );
```

When the playhead passes over this keyframe (wherever it may be on the timeline), the statement is executed. In this case, this command instructs the playhead to stop at this point on the timeline.



Figure 1.1 *The main splash screen of Flash MX Professional*

What is interesting is that ActionScript is like the English language. Most non-programmers will have understood the instruction to stop. Because of the similarity of script-based languages to natural language it should be quite easy to learn.

Over the past few versions of Macromedia Flash, the capabilities of the Flash Player have increased dramatically. As recently as Flash 3, for example, there was no ActionScript. Interactivity was limited to the most basic actions, such as stop and play.

Macromedia Flash 4 introduced the ActionScript scripting language with variables and loops. Macromedia Flash 5 added dot syntax, predefined objects such as Math, Sound and Date, String objects and an XML Parser. All of these are significant improvements in interactivity. ActionScript is based on the ECMA-262 specification that became the blueprint for JavaScript. Netscape was the originator of the JavaScript language, it was formerly known as LiveScript. Microsoft came along with their version of this language and named it Jscript. The differences between what should have been the same language meant that developers had to rewrite some of their script to make it more compatible with both IE and Netscape. The European Computer Manufacturers Association (ECMA) created the standard. ActionScript is based on the ECMA-262 specification, although it does not adhere to it totally.

ActionScript – The Language

Flash was born out of the lack of good animation capabilities for the web. With basic web capabilities limited to animated GIFs, Java applets, and later DHTML, Flash provides much better animation support by an animation framework, mixed media (including vector graphics, bitmaps with multiple levels of transparency, anti-alias text and sound) and a single cross-platform scripting model. Creating animated clips and movies in Flash is much easier than creating similar effects in DHTML: there is less coding involved and Macromedia controls the Flash plug-ins for various platforms, and making things work across different platforms is much easier. Support for vector graphics and the graphic transformation capabilities also decreases download size.

The Action Panel

Familiarize yourself with its features, because it's a panel you will use time and time again. You want as much room as possible to see your scripting, especially if you start to make long comments, so open it up as wide as possible. If you are using Normal Mode, all the syntax can be chosen from the “+” button at the top left-hand pane. Once you have a nice size window, don't keep opening and closing it: use its windowshade feature to collapse it into its title bar by double-clicking the title bar on Windows, and clicking the top right button on a Mac.

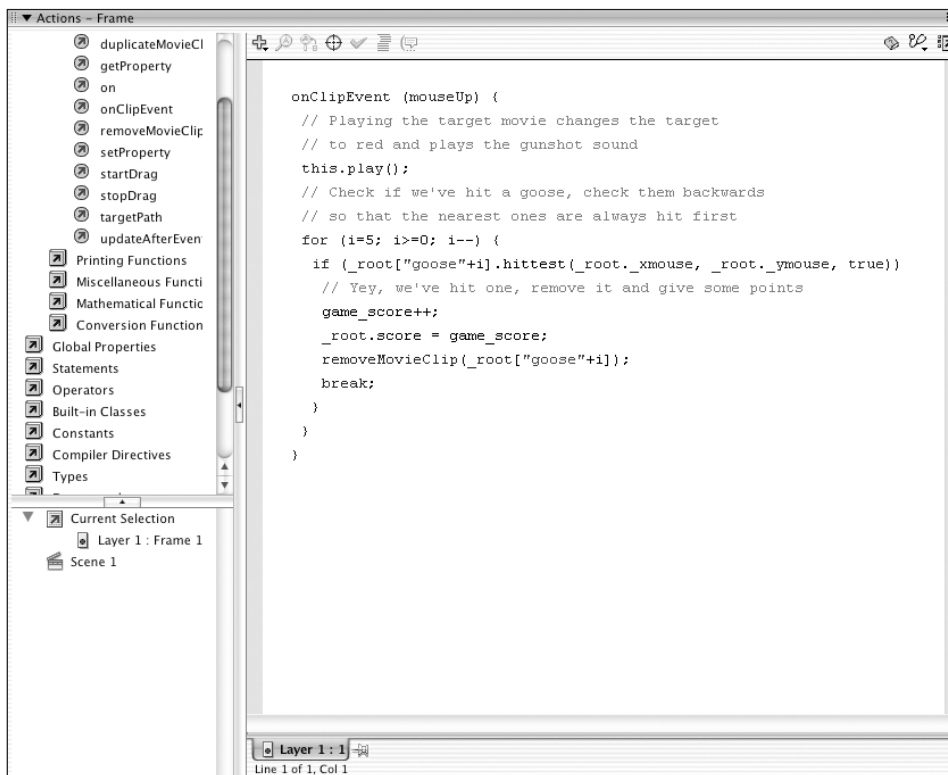


Figure 1.2 The Action panel

Using the Actions Panel

To add an action to a Flash document, you must attach it to a button or movie clip, or to a frame in the timeline. The Actions panel allows you to select, drag and drop, rearrange, and delete actions. You can use the Actions panel in two different editing modes: Normal and Expert. In Normal Mode you write actions by filling in parameter text boxes. Normal Mode is especially helpful for users who are new to ActionScript because it prevents most typos and syntax errors. However, experienced scripters will most likely find it easier and faster to use Expert Mode and simply type scripts directly. In Expert Mode you write and edit actions directly in a Script pane, much like writing scripts with a text editor.

To display the Actions panel, choose Window>Actions or press F2. To activate the Actions panel select an instance of a button, movie clip or frame, and the Actions panel title changes to reflect the selection.

Navigating Through the Actions Toolbox (see Figure 1.3)

To select the first item in the Actions toolbox, press Home on your keyboard. To select the last item in the Actions toolbox, press End on your keyboard. If you want to select the previous item in the Actions toolbox, press the Up Arrow or Left Arrow key. Press the Down Arrow or Right Arrow key to select the next item in the Actions toolbox, to expand or

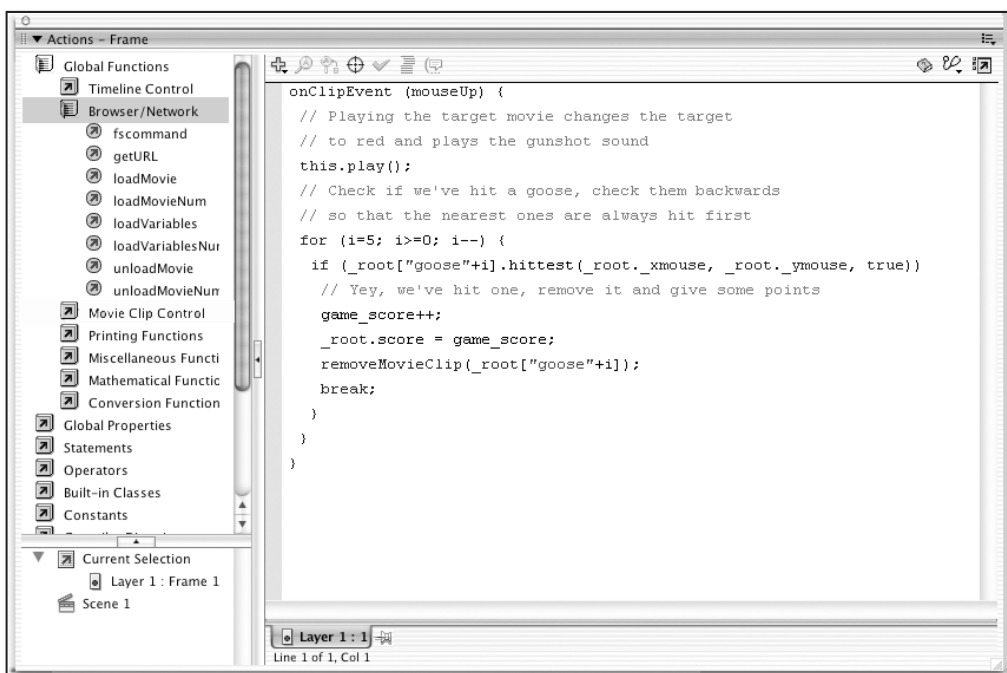


Figure 1.3 The toolbox area of the Actions panel

collapse any of these folders, press Enter or Spacebar. To insert an item into a script, press Enter or Spacebar.

To scroll up a page of items, press Page Up, to go down a page of items, press Page Down. To search for an Actions toolbox item by initial character, type the character, but remember this is not case sensitive. You can type the character many times to cycle through all the items that start with that character.

Viewing a Description of an Action

To view a description of an action, click a category in the Actions toolbox to display the actions in that category, and click an action. Select a line of code in the Script pane and the description appears at the upper right of the Actions panel. To add or delete actions in the Script pane, click a category in the Actions toolbox to display the actions in that category. Either double-click an action, drag it to the Script pane, or right-click (Windows) or Control-click (Macintosh) and select Add to Script. Click the Add (+) button and select an action from the pop-up menu. To delete an action, select a statement in the Script pane. Click the Delete (–) button or press the Delete key. To Move a statement up or down in the Script pane you need to select a statement in the Script pane, then click the Up or Down Arrow button or select the statement and drag it up or down.

Working with Parameters

This section displays nothing but gray until a line of code in the Script pane has been highlighted. For example, if you add the trace action to the Script pane and highlight it, an input field in the parameter fields section is brought up, prompting for a message value.

Searching for Text in a Script

To go to a specific line in a script, choose GoTo Line from the Actions panel pop-up menu; then enter the line number. To find text, click the Find button above the Script pane and choose Find from the Actions panel pop-up menu. Enter the text you want to find in the dialog box that appears. To find text again, press F3 or choose Find Again from the Actions panel pop-up menu.

To replace text, click the Replace button above the Script pane and choose Replace from the Actions panel pop-up menu. Enter the text you want to find and the text you want to replace it with in the dialog box that appears.

In Expert Mode, Replace scans the entire body of text in a script. In Normal Mode, Replace searches and replaces text only in the parameter box of each action. For example, in Normal Mode you cannot replace all gotoAndPlay actions with gotoAndStop.

You can navigate between scripts by using the jump menu at the top of the Actions panel and choosing a script from the list.

Pinning a Script to the Actions Panel

Click the Script Pin button. The Actions panel displays the script in the Script pane even when you click away from the object or frame.

Resizing the Actions Toolbox or Script Pane

Drag the vertical splitter bar that appears between the Actions toolbox and Script pane. You can also double-click the splitter bar to collapse the Actions toolbox; double-click the bar again to redisplay the Actions toolbox. Another way is to click the arrow button on the splitter bar to expand or collapse the Actions toolbox. When the Actions toolbox is hidden, you can still use the Add (+) button to access its items.

Viewing Line Numbers in the Script Pane

Select View Line Numbers from the View Options pop-up menu above the Script pane. Select View Line Numbers from the Actions panel pop-up menu or press Control+Shift+L (Windows) or Command+Shift+L (Macintosh).

Printing Actions

All the ActionScript from the Actions panel pop-up menu can be printed by choosing Print. The Print dialog box appears or choose Options and click Print.

The printed file will not include information about the source Flash file, so it's a good idea to include this information in a comment action in the script.

Checking Syntax

Click the Check Syntax button. Choose Check Syntax from the Actions panel pop-up menu. Syntax errors are listed in Output window.

Auto Formatting

ActionScript gives you flexibility in how you format your code. You can use any form of spaces, lines and indentations. In fact, if you are an experienced programmer you will probably find that you have a style that can easily be replicated by the auto formatting tool. In Normal Mode Flash enforces its own formatting; in Expert Mode it is up to you how you format your code. It is for this reason that creators of Flash have created the Auto Format button. Click the Auto Format button or choose Auto Format from the Actions panel pop-up menu. Then press Control+Shift+F (Windows) or Command+Shift+F (Macintosh). This functionality is useful if you are drawing on ActionScript from a number of different programmers.

The rules that the Auto Format feature follows are set in the Auto Format options dialog box. This can be accessed from the Actions panel pop-up menu (see Figure 1.4). In the dialog box you will find a number of check boxes from which you can alter the formatting style. Below the Options is a preview which allows you to test all the formatting styles.

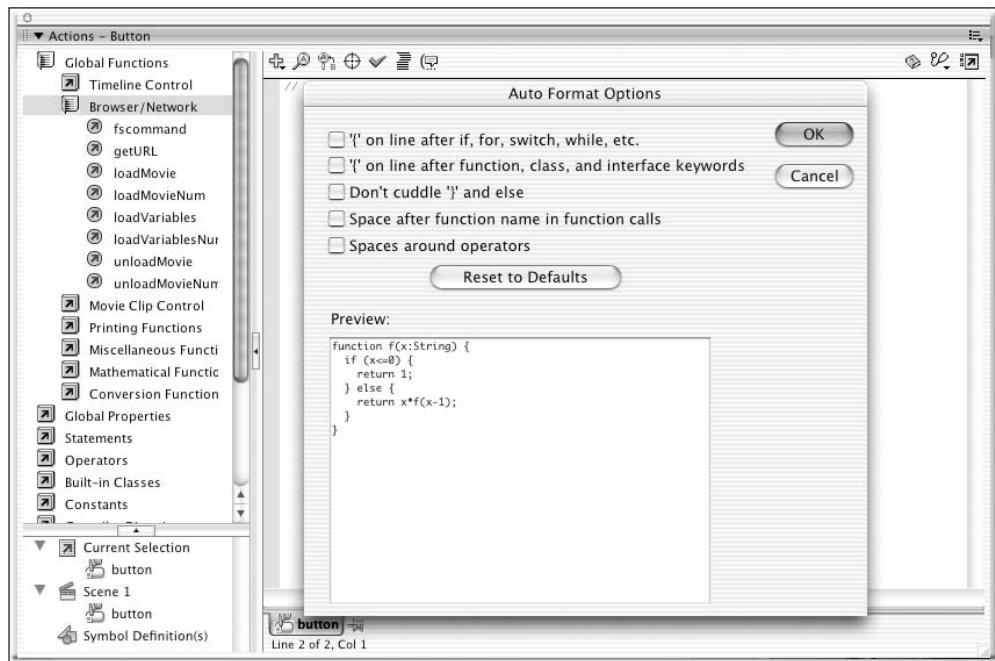


Figure 1.4 *Auto Format Options box*

Automatic indentation is turned on by default. To turn it off, deselect Automatic Indentation in ActionScript Editor preferences. When automatic indentation is turned on, the text you type after (or { is automatically indented according to the Tab Size setting in ActionScript Editor preferences. To indent another line in the Script pane, select the line and press Tab. To remove the indent, press Shift+Tab.

Using Punctuation Balance

Click between braces, brackets or parentheses in your script and press Control+['] (Windows) or Command+['] (Macintosh) to highlight the text between braces, brackets or parentheses. The highlighting helps you check whether opening punctuation has correct corresponding closing punctuation.

Switching Between Editing Modes

While working in the Actions panel, you can switch between Normal Mode and Expert Mode at any time. When you switch modes, Flash maintains your script's formatting unless you change the

script. For example, if you write a script in Expert Mode with your own style of indentation and switch to Normal Mode to view it, but make no changes, the formatting does not change. If, however, you modify the script in Normal Mode, Flash removes your custom indentation and formats the script using the Auto Format. Flash does warn you before changing any formatting.

To switch editing modes, do one of the following: choose Normal Mode or Expert Mode from the Actions panel pop-up menu or from the View Options pop-up menu above the Script pane. A check mark indicates the selected mode. To switch to Normal Mode, press Control+Shift+N (Windows) or Command+Shift+N (Macintosh). To switch to Expert Mode, press Control+Shift+E (Windows) or Command+Shift+E (Macintosh).

The Actions panel remains in the selected mode until you choose the other mode, even if you select a script on a different button, movie clip or frame. You cannot use Normal Mode to view an Expert Mode script that contains errors. If you try, you'll receive the message "This script contains syntax errors. It must be edited in Expert Mode."

You can use Normal Mode to view an Expert Mode script that uses ActionScript elements that are not supported by the current publish settings. If you export such a script, you'll receive a warning message.

Using the Reference Panel

You can use the Reference panel to view detailed descriptions of the actions listed in the Actions toolbox. The panel also displays sample code, which you can copy and paste into the Script pane of the Actions panel. You can also adjust the font size and print the contents of the Reference panel.

Viewing a description and sample code for an action you need to click an action in the Actions toolbox, then click the Reference button. Choose Window>Reference or press Control+Alt+Shift+F1 (Windows) or Command+Alt+Shift+F1 (Macintosh). In Normal Mode, select an action in the Script pane and click the Reference button at the upper right above the Script pane.

To print the contents of the Reference panel, select Print from either the context menu or the pop-up menu in the upper right of the Reference panel.

Copying and Pasting Sample Code

Highlight the code and right-click (Windows) or Control-click (Macintosh), then select Copy from the context menu. In the Script pane, right-click (Windows) or Control-click (Macintosh) and select Paste from the context menu or from the pop-up menu in the upper right.

Font Size

Select Large, Medium or Small Font from the pop-up menu in the upper right of the Reference panel.

Using an External Text Editor

You can use a text editor to write and edit ActionScript outside the Actions panel. You can export actions from the Actions panel to a text file, import a text file into the Actions panel, or use the include action to add an external script file at runtime.

In turn you can export actions as a text file: from the Actions panel pop-up menu (at the upper right of the panel), choose Export As File. Choose a location where the file will be saved, and click Save. You can then edit the file in an external text editor.

Importing a text file containing ActionScript can be done by the Actions panel pop-up menu, choosing the Import From File command. Select a text file containing ActionScript and click Open. Scripts with syntax errors can be imported only in Expert Mode. In Normal Mode, you'll receive an error message.

Externalizing ActionScript Code

ActionScript can call external text files which use the “.au” extension or “.swf” files. Maintaining the code in external files, you can standardize your code using libraries and therefore make the code easy to share. You can import external code into Flash using the import from file, the #include or the shared library.

It functions like an operator as it does not use parentheses to enclose its parameters. You must specify a location that can be either absolute, such as “c:\sprite.as”, or relative, as in “..\sprite.as”. The files must be local. So you cannot specify a non-local URL for the location of the “.as” file. The parameter, path and filename for the external file must always be enclosed in double quotation marks. You must not use a semicolon with this function as you will produce an error.

The “.as” extension is short for ActionScript. It is not necessary to name your files with this extension as you can use “.txt” but it makes standardizing code a whole lot easier if everybody is working to the same format.

Externalizing your code can be a good way of coping with complex movies; in reality it can also cause great confusion to the beginner, who may be unable to see what code goes with what objects. Working in this way will have a major effect on the way you work – not always for the better.

Adding an External Script to a Script within Flash when the Movie is Exported

1. Click in the Script pane to place the insertion point where you want the external script to be included.
2. In the Actions toolbox select the Actions category; then select the Miscellaneous Actions category.
3. Double-click the include action to add it to the Script pane.
4. Enter the path to the external file in the Path box.

The path should be relative to the FLA file. For example, suppose your FLA file is `sprite.fla` and your external script is called `externalfile.as`. If `sprite.fla` and `externalfile.as` are in the same folder, the path is `externalfile.as`. If `externalfile.as` is in a subfolder called `Scripts`, the path is `scripts/externalfile.as`.

The text of the external script replaces the `include` action when the document is exported as a Flash movie (SWF) file. The included code is only considered at “compile time”. In other words, the included “.as” files do not need to be staged/posted to the live site along with the SWFs.

Syntax Highlighting

In ActionScript, as in any language, syntax is the way elements are put together to create meaning. If you use incorrect ActionScript syntax, your scripts will not work.

In Normal Mode, syntax errors are red in the Script pane. If you move the mouse pointer over an action with incorrect syntax, a tooltip displays the associated error message; if you select the red highlighted action, the error message appears in the status bar at the bottom of the Actions panel. In both Expert and Normal Modes, ActionScript export version incompatibilities are yellow in the Script pane. For example, if the Flash Player export version is set to Flash 5, ActionScript that is supported only by Flash Player 6 is yellow.

Using Code Hints

Flash can detect what action you are entering and display a code hint – a tooltip containing the complete syntax for that action or a pop-up menu listing possible method or property names. In Expert Mode, code hints appear for parameters, properties and events when you enter certain characters in the Script pane. In Normal Mode, code hints appear for parameters and properties, but not events. They appear in the parameter text boxes when the Expression box is selected. Code hints are enabled by default. By setting preferences, you can disable code hints or determine how quickly they appear. When code hints are disabled in preferences, you can turn them on manually.

Code Hints and Tooltip-style Hints

Click the Show Code Hint button so that code hints always appear. To enable automatic code hints choose Preferences from the Actions panel pop-up menu. On the ActionScript Editor tab, select Code Hints.

For tooltip-style hints, type an open parenthesis `[` after an action name. The code hint appears. Enter a value for the parameter. If there is more than one parameter, separate the values with commas. To dismiss the code hint, do one of the following:

- Type a closing parenthesis `]`.
- Click outside the statement.
- Press Escape.

You can have manual code hints in Expert Mode by clicking the Show Code Hint button above the Script pane and from the Actions panel pop-up menu by choosing Show Code Hints.

The menu-style code hints display by doing one of the following:

- Type a dot after the suffix of an object name.
- Type an open parenthesis `[(]` after an event handler name.
- To navigate through the code hint, use the Up and Down Arrow keys.
- To select an item in the menu, press Return or Tab, or double-click the item.

To dismiss the code hint, do one of the following:

- Choose one of the menu items.
- Click outside the statement.
- Type a closing parenthesis `[)]` if you've already typed an open parenthesis.
- Press Escape.

Many ActionScript objects require you to create a new instance of the object in order to use its methods and properties. In a piece of code like `myMc.gotoAndPlay(6)`, the `gotoAndPlay` method tells the instance `myMc` to go to a certain frame and begin playing the movie clip. The Actions panel doesn't know which code hints to display and what type of object the instance `myMc` is.

You can display code hints that help you choose the correct syntax and structure for your code, but you must add a special class suffix to each instance name. To display code hints for the class `MovieClip`, you must name all `MovieClip` objects with the suffix `"_mc"`, as in the following examples:

```
Car_mc.gotoAndPlay(1);

Ball_mc.stop( );

Stamp_mc.duplicateMovieClip('NewStamp_mc', 100);
```

Table 1.1 shows the suffixes and their corresponding object classes.

You can also use ActionScript comments to specify an object's class for code hinting. The following example tells ActionScript that the class of the instance `theObject` is an `Object`. If you were to enter the code `"mc"` after these comments, a code hint would display the list of `MovieClip` methods and properties; if you were to enter the code `"theArray"`, a code hint would display a list of `Array` methods and properties.

```
// Object theObject;
// Array theArray;
// MovieClip mc;
```


Table 1.1 *Suffixes and object classes*

Suffix	Object class
_mc	MovieClip
_array	Array
_str	String
_btn	Button
_txt	TextField
_fmt	TextFormat
_date	Date
_sound	Sound
_xml	XML
_xmlsocket	XMLSocket
_color	Color
_camera	Camera
_mic	Microphone
_stream	NetStream
_connection	NetConnection
_so	SharedObject
_video	Video

Setting Actions Panel Preferences

To set preferences for the Actions panel, you use the ActionScript Editor section of Flash preferences. You can change settings such as indentation, code hints font and syntax coloring, or restore the settings to their defaults. To set Actions panel preferences you need to choose Preferences from the Actions panel pop-up menu, then choose Edit>Preferences and click the ActionScript Editor tab.

You can set any of the following preferences: for Editing Options, select Automatic Indentation to automatically indent ActionScript in the Script pane in Expert Mode, and enter an integer in the Tab Size box to set an indentation tab size for Expert Mode (the default is 4). Select Code Hints to turn on syntax, method and event completion tips in both Expert and Normal Modes. Move the Delay slider to set the amount of time Flash waits before displaying a code hint (the default is 0).

For Text, select a font or size from the pop-up menu to change the appearance of text in the Script pane.

For Syntax Coloring, choose a color for the Script pane's foreground and background and for keywords (for example, new, if, while and on), built-in identifiers (for example, play, stop and gotoAndPlay), comments and strings.

To restore the default ActionScript Editor, click the Reset to Defaults button.

Movie Explorer

It used to be impossible to find scripts hidden within movie clips until the advent of the Movie Explorer. Using the row of buttons along the top of the window you can activate only the scripts

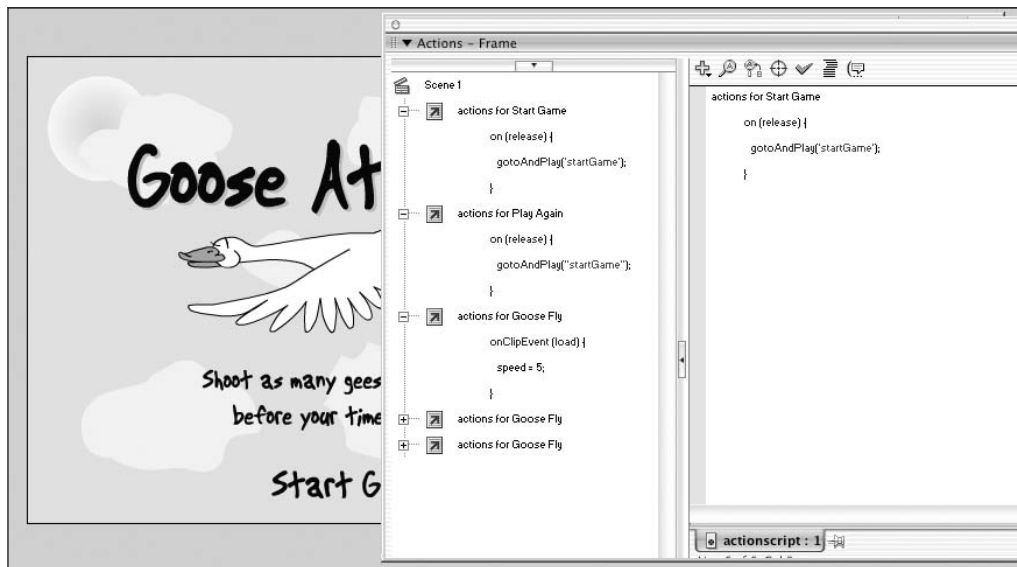


Figure 1.5 *The Movie Explorer window with only ActionScript selected*

view. You'll see in the list all the scripts you've written, and probably a few that you thought had been lost forever. Typing a keyword in the Find box further narrows the list of scripts, making it even easier to find. If someone's determined to hide their code there are lots of places in Flash to do so. The most useful and the only tool for detecting code at the limits is the Movie Explorer. Select Window>Other Panels>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 MovieClip. This same window will allow you to search for text fields containing a certain word. There is no easy way to search the entire movie but you can search one scene at a time. To filter the display to show only scripts, select the Show Actions Scripts icon under the Show menu, and deselect all the other icons.

Buttons

A button has the ability to change its image according to what state the button is in. There are four of these states, Up, Over, Down and Hit. You see the button in its normal state; you can then interact with it by rolling the mouse cursor "over" the button. When you click the button it will send it into the "Down" state for as long as the mouse button remains pressed. A button can also carry out an action when a certain state is reached. There is a four-framed timeline to define each state.

Creating a Button

Choose Insert>New Symbol and select "Button" as the behavior option. Once you click OK, Flash will enter into edit symbol mode. The timeline will change to show four blank frames corresponding to the four button states. The first frame will have a blank keyframe in it. Notice

how the button symbol is different to the graphic symbol. Each frame in the timeline of a button symbol has a specific function:

- **Up** is the state used whenever the mouse pointer is not over the button.
- **Over** is used when the mouse pointer is over the button.
- **Down** is the state attained when the button is clicked.
- **Hit** is invisible and defines the area where the button will react to the mouse pointer.

It is important to note that you can't use a button in a button but you can use a movie clip for an animated button or a graphic symbol. For the purpose of this tutorial we'll create a circle. Place your circle on the stage and you'll notice that the "Up" frame now has a full keyframe in it. Now click in the second frame, labeled "Over", and choose Insert>Keyframe. The circle image from the first frame will appear on the Stage. This frame is essentially the rollover function, so you'll want the image to be different. Modify the circle image by placing a smaller circle of a different color in the center. When the mouse is over the button this circle will appear. Try it.

Repeat this step for the "Down" and "Hit" frames. Keep your "Down" image the same as "Up" but nudge it slightly right and down the page from the "Up" position. This will give an impression that the button is pressed when clicked on. To preview your button in editing mode choose Control>Enable Simple Buttons and you can see the button animate when you press it.

The "Hit" frame defines the area that will react to the mouse. It can be as big or small as you like, but it's best to ensure that the frame contains a solid shape that is large enough to surround all the images you have placed in the button's frames. In terms of the circle, you want the whole circle to interact with the mouse, so a copy of the circle image in this state will suffice. The objects in the "Up" frame will be used as the "Hit" frame if you do not specify a "Hit" state.

Frame Actions

The event handler for a keyframe is implied by definition, and you will never write code to specify the handler because as the playhead enters a new frame, an event occurs.

Jumping to a Frame or Scene

To jump to a specific frame or scene in the movie, you use the Go To action. When the movie jumps to a frame, you can play the movie from the new frame (the default) or stop at the frame. The movie can also jump to a scene and play a specified frame or the first frame of the next or previous scene.

To jump to a frame or scene:

1. Create a new file and name it Goto.fla.
2. Create a button in the middle of the screen.
3. Choose Window>Development Panels>Actions to display the Actions panel.

4. From the Actions list, select the Actions>Movie Control>goto action. Flash inserts the gotoAndPlay action in the Actions list.
5. Create a Scene 2. To keep playing the movie after the jump, leave the gotoAndPlay option (the default) selected in the Parameters pane. To stop the movie at a specified frame, select gotoAndStop. The action changes to gotoAndStop. Leave the gotoAndPlay option selected. When you follow these directions in Expert Mode you will get the error (as expected) that the gotoAndPlay action must appear within an on() handler. Since Expert Mode does not automatically insert the handler code, it should be noted that the instructions assume that Normal Mode is selected.
6. In the Scene pop-up menu in the Parameters pane, specify the destination scene, in this case it is Scene 2, or you can select <next scene>.
7. Leave the Type pop-up menu set at Frame Number and leave the number at 1.
8. Now enable simple buttons in the work area (Control>Enable Simple Buttons) and click on the button in Scene 1; you will be taken to Scene 2.

Next or Previous Frame, Frame Number, Frame Label or Expression allow you to specify a frame. Expressions are any part of a statement that produces a value, such as 1+1.

If you chose Frame Number, Frame Label or Expression, enter the frame by number, label, or an expression that evaluates to a frame number or label. This example goes to Scene 1, frame 5: gotoAndStop(“Scene 1”, 5). The following statement indicates the frame that is five frames ahead of the frame that contains the action: gotoAndStop(_currentframe+5);

Let’s try putting an event handler in a movie clip.

1. Create a new movie.
2. Draw a circle on the stage.
3. Select the circle and then go to Insert>Convert to Symbol.
4. Name the symbol circle and select movie clip as the behavior.
5. Click OK.
6. Select the circle movie clip.
7. Open the Actions panel window by going to Window>Actions.
8. Type the following (in Expert Mode):

```
onClipEvent(keyDown){
    This._visible = false;
}
onClipEvent(keyUp){
    This._visible = True;
}
```

In this code, pressing down makes the object invisible, releasing the key makes it visible again. Select Control>Test Movie. If you keydown on any of the keys on your keyboard you’ll notice the circle becomes invisible.

The following is two simple event handlers, one attached to a button and the other attached to a MovieClip.

1. Create a new movie.
2. Drag a button from Window>Common Libraries>Buttons.
3. Select the button and then open the Actions panel window by going to Window>Actions.
4. Type in Expert Mode:

```
on(release){  
    trace('button clicked');  
}
```

5. Select Control>Movie.
6. Click the button and you will see the message “button clicked” appear in a new window.

So what happened? When the movie plays and you click the button the release event is detected by the Flash Player which executes the on(release) event handler. The result is the trace opens a window with our message in it “button clicked”.

Where Does the ActionScript Go?

There are three places you can assign actions: on a keyframe, within an on() event handler or within an onClipEvent() handler. Actions added to these places are Frame actions, Button actions and MovieClip actions, respectively.

Dot Syntax

The dot syntax object-oriented syntax is used to reference methods and properties of different objects and movie clips in your movie from ActionScript. Take, as an example, the getting and setting of properties. The properties of movie clips can now be accessed much more quickly. Let's say you want to find the position of a movie clip called “Car” and move it 10 pixels to the right. This can now be easily done using

```
Car._x = Car._x + 10
```

Or if we wanted to play the Car movie from frame 2, we would use the syntax

```
Car.gotoAndPlay(2);
```

You can also use the dot syntax to access movie clips that are embedded within other movies. For example, if the Car movie contained a movie called “Wheel”, you could play the wheel movie using

```
Car.Wheel.play();
```

Or you could move the wheel down by using

```
Car.Wheel._y = Car.Wheel._y + 10;
```

As well as accessing the Wheel movie from the root timeline, you can also access any movie from any other movie by using the “_parent” and “_root” objects. For example, if we had a script in the Wheel movie that needed to set a change in a variable on the Car movie, we would use the syntax:

```
_parent._x = _parent._x - 10;
```

Likewise, if the Car movie contained a script that needed to change a variable on the root timeline it could use the syntax:

```
_parent.rootVar = 123;
```

The “_parent” movie can also be embedded to descend further down the path. For example, to access the root timeline from the Wheel movie, you would use the syntax:

```
_parent._parent.rootVar = 321;
```

Using the “_parent” syntax is referred to as being a relative path – that is, the path is relative to the calling movie clip. An absolute link is constructed from the base root timeline. For example, an absolute path from the Wheel movie, or any other movie clip, to the Car movie would be:

```
_root.Car._x = _root.Car._x - 10;
```

You will come across tutorials later in the book that explain the dot syntax in more depth, so do not worry if you have not got the hang of them yet, this is only Chapter 1 after all.

2 Constructing and Sharing ActionScript

Loading and Unloading Movies

To play additional movies without closing the Flash Player, or to switch movies without loading another HTML document, use the Load Movie action. The Unload Movie action removes a movie previously loaded by the Load Movie action. Not only can it be used to incrementally load sections of large Flash projects, but it can be used to manage code revisions by loading SWFs that only contain code.

To load a movie:

1. Create a new file and place a button in the middle of it. Select the button and choose Window>Development Panels>Actions to display the Actions panel. This example assumes Normal Mode is selected.
2. In the Toolbox list, click the Actions>Browser/Network category to display the web browser and network actions, and select the Load Movie action. In the Parameters pane, for URL type movie2.swf.
3. Save it as movie1 fla.
4. Create a file with a green square in the middle of it and save it as movie2 green; movie2 fla.
5. Export movie2 fla as movie2.swf and place it in the same folder as movie1 fla.
6. For use in the Flash Player or for testing in Flash, all the SWF files must be stored in the same folder and listed as file names without folder or disk drive specifications. This makes testing convenient, but it is fine to specify absolute and relative paths to other folders/directories.
7. Now test the movie, Control>Test Movie, and click the button in the middle of the screen; movie2 will load into movie1.

If you choose Level for Location, enter a level number as follows:

- To load the new movie in addition to existing movies, enter a level number that is not occupied by another movie.
- To replace an existing movie with the loaded movie, enter a level number that is currently occupied by another movie.
- To replace the original movie and unload every level, load a new movie into level 0. The movie loaded first is loaded at the bottom level.
- The movie in level 0 sets the frame rate, background color and frame size for all other loaded movies. Movies may then be stacked in levels above the movie in level 0.

To unload a movie from a Flash movie window:

1. Select the frame, button instance or movie clip that you will assign the action to. Choose Window>Actions to display the Actions panel.
2. In the Toolbox list, select the Actions>Browser/Network category to display the web browser and network actions, and select the Unload Movie action. For Location, choose one of the following options from the pop-up menu: for a loaded movie, select Level and enter the level of the movie that you want to unload.
3. To target a movie to unload, select Target and enter the path of the movie that you'll target to unload. To enter an expression that evaluates to a level or movie, select Expression and enter the expression. For example, "unloadMovie(3);" targets the movie on level 3 and unloads it.

To test a Load Movie or Unload Movie action, make sure that the movie being loaded is at the specified path. If the path is an absolute URL, an active network connection is required. The Load Movie and Unload Movie actions do not work in editing mode. You need to build the SWF to see them work. Choose Control>Test Movie.

Movie Clips and Buttons Overview

A movie clip is like a mini-movie: it has its own timeline and properties and can be added to any timeline as a separate object. The symbol for a movie clip can be used several times in a Flash document. This is accomplished by dragging the movie clip out of the library; each time you drag a copy out of the library you are in fact dragging out an instance of the movie clip. If you want to reference these instances in ActionScript then you must give each instance a unique name to distinguish instances from each other. As you start to make complex movies you will start to depend on nesting movie clips inside each other to create a hierarchy. The result is a hierarchy of movie clips.

Nested movies form a hierarchy, or a tree, with each movie clip occupying a position or "node" in the tree. Each clip has its own timeline. This hierarchical tree appears as a display list. Movies that are loaded into the Flash Player with the loadMovie action also have independent timelines and a position in the display list. Movie clips are controlled using actions, which in turn invoke movie clip methods. To control a specific movie clip, its target path must be determined. A movie clip's target path indicates its position in the movie's hierarchy. A clip called "clip1" on the main timeline would have a target path like `_root.clip1`. If there were a clip inside clip1 called "clip2", it would have a path like `_root.clip1.clip2`. There will be more on target paths in upcoming chapters. There are many powerful movie clip methods including ones to: drag a movie clip, dynamically add a movie clip to a scene, turn a movie clip into a mask, and draw lines and fills on the stage, etc.

Just like each movie clip instance, each button instance is an ActionScript object with its own properties and methods. You can give a button an instance name and manipulate it with

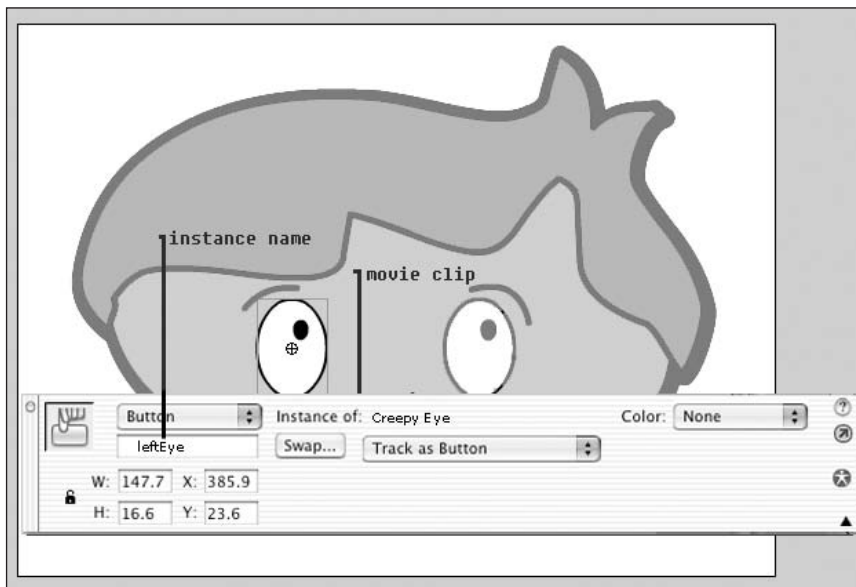


Figure 2.1 Properties dialog box showing the instance name of a movie clip



Figure 2.2 The Movie Explorer shows the display list

ActionScript. Each movie clip and button in a Flash document are objects with properties and methods that can be changed by ActionScript.

About Multiple Timelines

The Flash Player has a stacking order of levels. Every Flash movie has a main timeline located at level 0 in the Flash Player. This governs the overlapping not of instances, but of “.swf” files loaded into the player via the `loadMovie()`. You can use the `loadMovie` action to load other Flash movies (SWF files) into the Flash Player at different levels. If you load movies into levels above level 0, the movies lie on top of each other like drawings on transparent paper; where there is no content on the stage, you can see through to the content on lower levels. If you load a movie into level 0, it replaces the main timeline. Each movie loaded into a level of the Flash Player has its own timeline. If you load an “.swf” file into an occupied level; that level’s previous occupant is replaced by the newly loaded document.

Flash movies at any level can have movie clip instances on their timelines. As explained above, each movie clip instance also has a timeline and can contain other movie clips that also have timelines. When you author in Flash, you can view the display list in the Movie Explorer; when you play the movie in test mode, the stand-alone Flash Player or a web browser, you can view the display list in the Debugger.

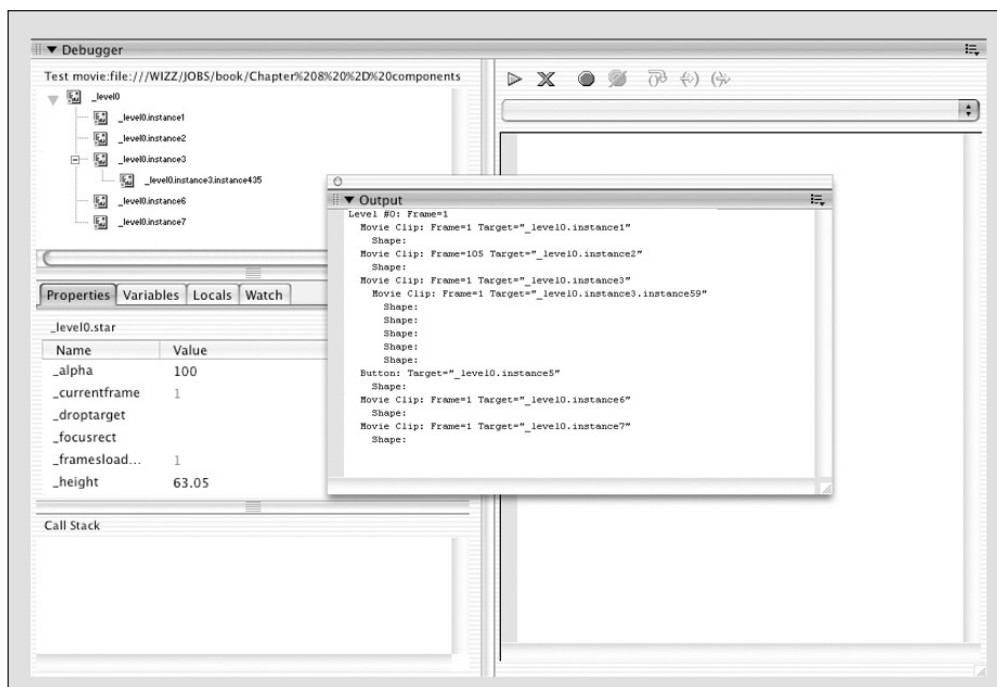


Figure 2.3 Levels viewed in the Debugger

Depending on their locations in the display list, timelines have specific relationships with each other. A child timeline nested inside another timeline is affected by changes made to the parent timeline. So scaling a movie clip of a head will also scale up the eyes, assuming that they are nested in the head movie clip timeline.

Movie Clip Hierarchy

When you place a movie clip instance on another movie clip's timeline, the placed movie clip is the child and the other movie clip is the parent. The parent instance contains the child instance. The root timeline for each level is the parent of all the movie clips on its level, and because it is the topmost timeline, it has no parent. The parent-child relationships of movie clips are hierarchical. You can use the movie clip hierarchy in Flash to organize related visual objects. Any change you make to a parent movie clip is also performed on its children.

Absolute and Relative Target Paths

You can use actions to send messages from one timeline to another. The timeline that contains the action is called the controlling timeline, and the timeline that receives the action is called the target timeline. For example, there could be an action on the last frame of one timeline that tells another timeline to play. To refer to a target timeline, you must use a target path, which indicates the location of a movie clip in the display list.

Movie Reference Example

This example demonstrates how to use ActionScript to communicate between different timelines. The example features three input text fields in different movie clips with buttons to pass the input text values between the movie clips. The input panel contains a text field and a button; the button passes the value in the text field to the output panel. The output panel also contains a text field and a button to pass the value to the input panel. The center button passes the values of both text fields to the bottom text field.

Download www.sprite.net/understanding/movieref.fla; this contains the structure for the example. Double-click the input panel to edit it. Using the text tool create a new text field on the panel; create the text field on the “Text Field” layer. Select the text field and, on the Properties panel, select “Input Text” from the drop-down and set the value for the “Var” to “myInputText”. This will be the variable we use to manipulate this text field.

Go back to the root movie and double-click the output panel. Repeat the process, creating a new text field on the output panel. Again, select the text field, set its mode to “Input Text” and “Var” to “myOutputText”. Go back to the root movie, select the input panel and, on the Properties panel, name the movie clip “inputPanel”; this will be the name we use to access the input panel. Next, select the output panel and name this movie clip “outputPanel” on the properties panel. Edit the input panel again and open the actions panel of the button on the “Button” layer. To

copy the text from the text field on the input layer to the text field on the output panel, add the following code to the button:

```
on(release) {  
    _parent.outputPanel.outputText = inputText;  
}
```

This code uses a relative path to set the “outputText” text field on the “outputPanel” movie to the value in the “inputText” text field.

Now we have to do the same for the output panel. Go back to the root movie and edit the output panel. Select the button and open the Actions panel. Add the following code:

```
on(release) {  
    _root.inputPanel.inputText = outputText;  
}
```

This code uses an absolute path to set the “inputText” text field on the “inputPanel” movie to the value in the “outputText” text field. By running the movie at this point you will see that by entering text onto one panel and clicking the button will copy the text to the other panel. Next, go to the root timeline and create a text field at the bottom of the movie on the “Text Field” layer. On the properties panel, set its type to “Dynamic Text” and set the “Var” value to “rootText”. This will be used to display the concatenated string when the center button is clicked. The final step is to add the code to copy both text field values to the root text field. Select the button and add the following code:

```
on(release) {  
    rootText = inputPanel.inputText + ' ' +  
        outputPanel.outputText;  
}
```

This code will join the two text field values and place them in the root text field.

A complete version of this example can be found in `movieref_finished fla` (www.sprite.net/understanding).

A relative path depends on the relationship between the controlling timeline and the target timeline. Relative paths can address targets only within their own level of the Flash Player. For example, you can't use a relative path in an action on `_level0` that targets a timeline on `_level5`.

In a relative path, use the keyword `this` to refer to the current timeline in the current level; use the alias `_parent` to indicate the parent timeline of the current timeline. You can use the `_parent`

alias repeatedly to go up one level in the movie clip hierarchy within the same level of the Flash Player. For example, `_parent._parent` controls a movie clip up two levels in the hierarchy. The topmost timeline at any level in the Flash Player is the only timeline with a `_parent` value that is undefined.

Relative paths are useful for reusing scripts. For example, you could attach a script to a movie clip that enlarges its parent by 150% as follows:

```
onClipEvent (load) {  
    _parent._xscale = 150;  
    _parent._yscale = 150;  
}
```

You could then reuse this script by attaching it to any movie clip instance. Whether you use an absolute or relative path, you identify a variable on a timeline or a property of an object with a dot (.) followed by the name of the variable or property.

Reusability and Modular coding involves developing the framework and a consistency in a way that code can be written once in a particular file and called multiple times in different areas of an application. Rather than rewriting 60 lines of code, you can write an open-ended file that can handle an attribute passed to it, or wrap the process into a function.

Shared Library Assets

Allow a Flash movie to share library assets with other Flash documents, while authoring, or when a movie is played with the Flash Player. Shared runtime libraries help you create smaller files and easily make updates to multiple documents simultaneously by letting your document show library symbols and shared objects that are stored on an intranet or the Internet. You can improve your work pace by letting yourself track, update and swap symbols in any Flash document available on your computer or network.

Using Shared Library Assets

Shared library assets enable you to use an asset movie to feed content like logos and pictures to multiple destination movies. You can do this in two different ways:

Runtime shared assets

Assets from a source movie are linked as external files in a destination movie. These assets are loaded into the destination movie during movie playback at runtime. The source movie containing the shared asset does not need to be available on your local network when you author the destination movie. The destination movie must be referencing the content in the source movie at runtime.

Author-time shared assets

You can update or replace any symbol in a movie you are authoring, with any other symbol on your local network. The symbol in the destination movie retains its original name and properties, but its contents are updated or replaced with those of the symbol you select, which you may have updated through the source file.

Using shared library assets can optimize your workflow and movie asset management in numerous ways. We will go through an example demonstrating how you can use shared library assets to share a font symbol across multiple sites, providing a single source for elements in animations used across multiple scenes or movies. The shared library is best used as a central resource library to use for tracking and controlling revisions.

How Runtime Shared Assets Work

Two processes are involved in working with shared library assets:

1. You as the author must define a shared asset in the source movie, and enter an identifier label for the asset and a URL where the source movie will be posted.
2. The author of the destination movie defines a shared asset in the destination movie and enters an identifier string and URL identical to those used for the shared asset in the source movie. You can drag the shared assets from the posted source movie into the destination movie library. Whichever approach you take, the source movie must be posted to the specified URL in order for the shared assets to be available for the destination movie.

Defining Assets in a Source Movie

You use the Symbol Properties dialog box or the Linkage Properties dialog box to define sharing properties for an asset in a source movie, to make the asset accessible for linking to destination movies. The following example takes you through the stages:

1. Open the source movie and choose Window>Library to display the Library panel.
2. Select a movie clip, button or graphic symbol in the Library panel and choose Properties from the Library options menu. Click the Advanced button to expand the Properties dialog box.
3. For Linkage, select Export for Runtime Sharing to make the asset available for linking to the destination movie.
4. Enter an identifier for the symbol in the Identifier text field. Do not include spaces. This is the name Flash will use in identifying the asset when linking to the destination movie. The Linkage Identifier is also used by Flash to identify a movie clip or button that is used as an object in ActionScript.
5. Enter the URL where the SWF file containing the shared asset will be posted.
6. Click OK.

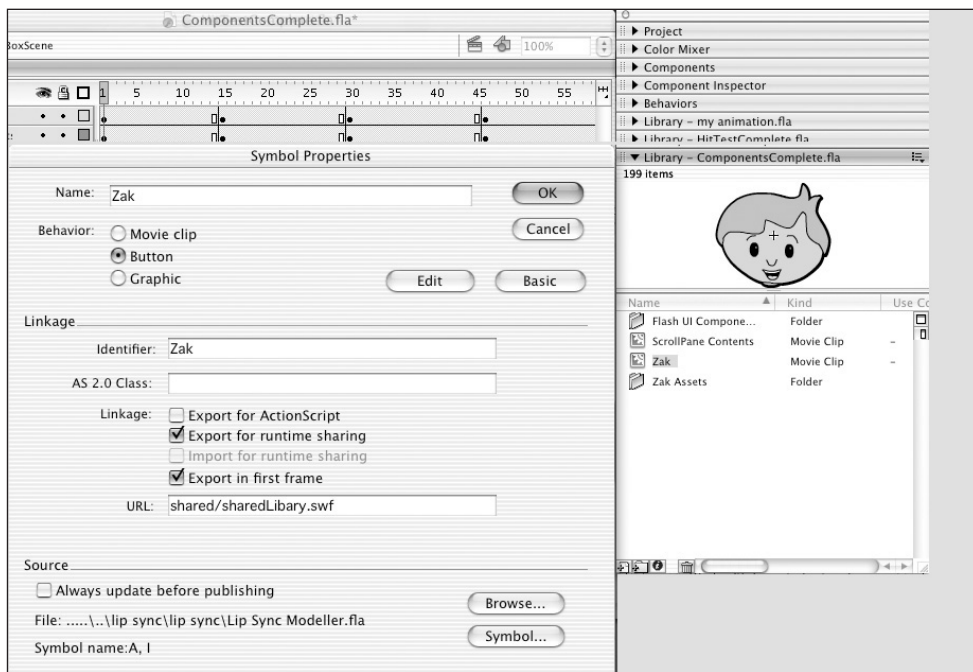


Figure 2.4 *The Linking Properties dialog box*

The URL value is a relative path; all movies that use this asset must be located in the directory above the shared folder. If the movies are located across several directories you need to use an absolute URL (<http://www.sprite.net/shared/sharedLib.swf>). If you specify a path as `/shared/sharedLib.swf` this specifies a shared folder located at the root of the current URL used by the destination Flash Movie. In the above example, when you publish the movie, you must post the SWF file to the URL specified in step 5, so that the shared assets will be available to destination movies.

Linking to Shared Assets from a Destination Movie

The Symbol Properties dialog box or the Linkage Properties dialog box is used to define sharing properties for an asset in a destination movie, to link the asset to a shared asset in a source movie. If the source movie is posted to a URL, you can also link a shared asset to a destination movie by dragging the asset from the source movie to the destination movie. You can turn off sharing for a shared asset in the destination movie; this allows you to embed the symbol in the destination movie.

Linking a shared asset to a destination movie by entering the identifier and URL is done as follows:

1. In the destination movie, choose **Window>Library** to display the Library panel.
2. Select a movie clip, button or graphic symbol in the Library panel and choose **Properties** from the Library options menu. Click the **Advanced** button to expand the Properties dialog box.

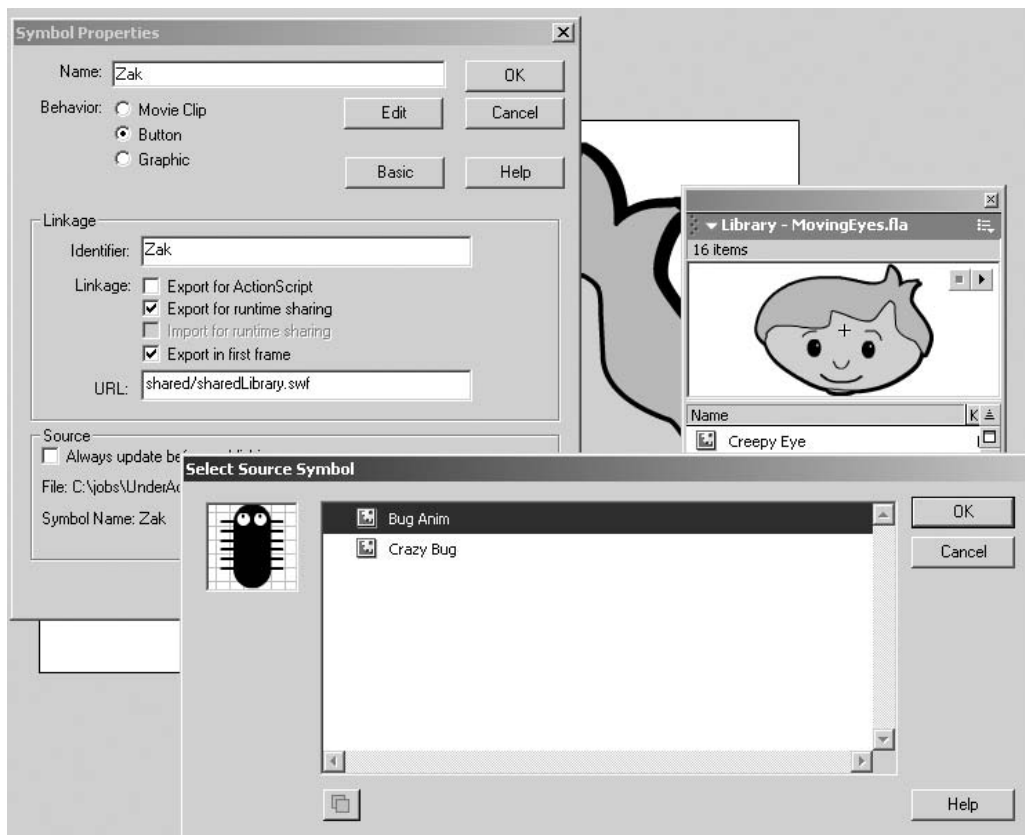


Figure 2.5 *Source Symbol dialog window*

3. Select a graphic symbol and choose Linkage from the Library options menu.
4. For Linkage, select Import for Runtime Sharing to link to the asset in the source movie.
5. Enter an identifier/name for the symbol in the text field that is identical to the identifier used for the symbol in the source movie.
6. Enter the URL where the SWF source file containing the shared asset is posted.
7. Click OK.

Linking a shared asset to a destination movie by dragging is carried out as follows:

1. In the destination movie, choose File>Open or Open as Library.
2. In the Open or Open as Library dialog box, select the source movie and click Open.
3. Drag the shared asset from the source movie Library panel into the Library panel or onto the stage in the destination movie.

To turn off linkage for a symbol in a destination movie:

1. In the destination movie, select the linked symbol in the Library panel and do one of the following: if the asset is a movie clip, button or graphic symbol, choose Properties from the

Library options menu. If the asset is a font symbol, choose Linkage from the Library options menu.

2. In the Symbol Properties dialog box or the Linkage Properties dialog box, deselect Import for Runtime Sharing.
3. Click OK.

Updating or Replacing Symbols using Author-time Sharing

You can update or replace a movie clip, button or graphic symbol in a movie with any other symbol in an FLA file accessible on your local network. The original name and properties of the symbol in the destination movie are preserved, but the contents of the symbol are replaced with the contents of the symbol you select. Any assets that the selected symbol uses are also copied into the destination movie.

To update or replace a symbol, proceed as follows:

1. With the movie open, select movie clip, button or graphic symbol and choose Properties from the Library options menu.
2. Select a new FLA file, under Source in the Symbol Properties dialog box, and click Browse.
3. In the Open dialog box, navigate to an FLA file containing the symbol that will be used to update or replace the selected symbol in the Library panel, and click Open.
4. To select a new symbol in the FLA file, under Source, click Symbol.
5. Navigate to a symbol and click Open.
6. In the Symbol Properties dialog box, under Source, select Always Update Before Publishing to automatically update the asset if a new version is found at the specified source location.
7. Click OK to close the Symbol Properties or Linkage Properties dialog box Shared library (runtime import).

Shared Libraries for Font Management

One of the most effective Shared Library assets is a Font symbol. Fonts are ideal shared assets as they can be quite heavy. In the following exercise a Font symbol will be shared by two other Flash movies:

1. Create a new Flash document and save it as sharedlib fla, placing it into a folder named shared.
2. Open the Library panel. From the top right of the panel select the Options menu and choose New Font. Select Trebuchet and name it trebuchet Shared, as in Figure 2.6.

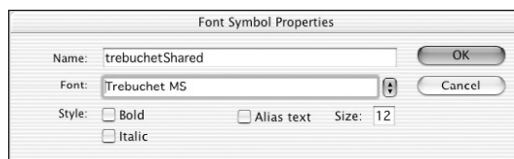


Figure 2.6 The Font Symbol Properties dialog box

3. Right-click (Control+click on a Mac) the `trebuchetShared` in the Library panel and choose the Linkage Properties dialog box. Select the “Export for ActionScript” and “Export for runtime sharing” check boxes. The identifier field should be name `trebuchetShared`. In the URL field, type in `shared/sharedLib.swf`.
4. Click OK.

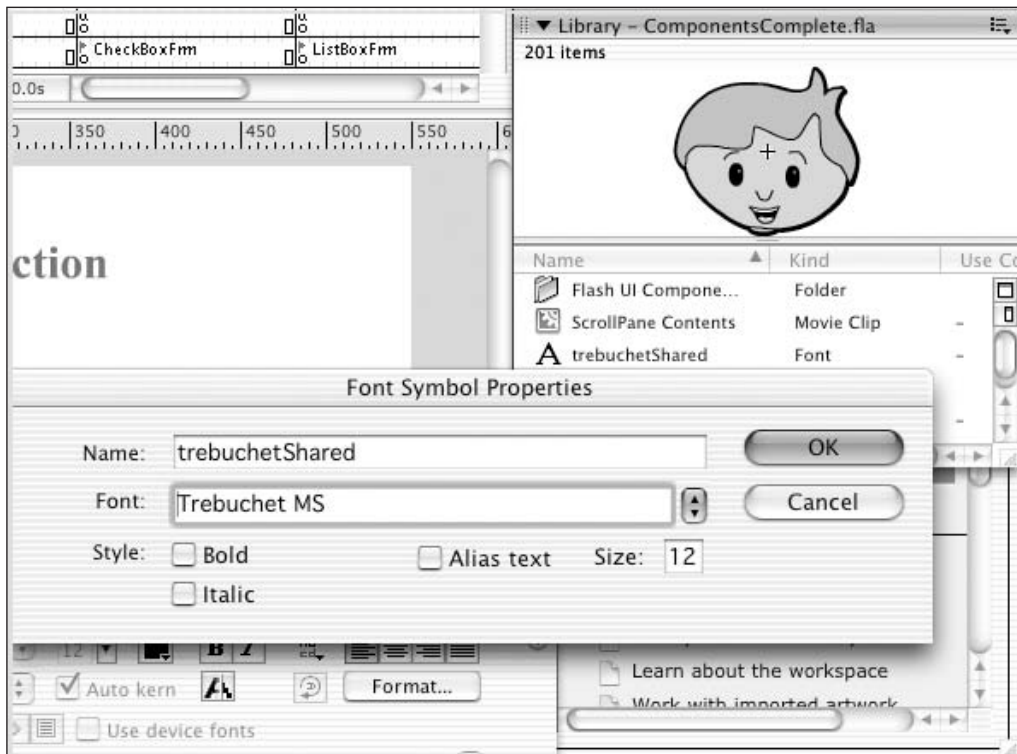


Figure 2.7 *The Linkage Properties dialog box*

5. Open the Publish Settings dialog box (File>Publish Settings). Disable the default names and the html check box. This will make sure that the Flash movie will not be renamed. Make sure that the final Flash Movie (.swf file) retains the name specified in the URL field “`sharedLib.swf`”.
6. Close the `sharedLib.fla` document, and create a new document. Save this file as `demo_shared.fla` in the folder above the `sharedLib.swf` file.
7. Select File>Open as Library, and select the `sharedLib.fla` file inside the shared folder. Drag the `trebuchetShared` symbol from the library to the stage of the `demo_shared.fla`. Close the Library panel.

8. Create a new layer and name it text. In frame 1 of this layer create a static text block and put some text into the field. You can now select the font `trebuchetShared` from the Property Inspector. All the shared fonts will have an asterisk (*) after their name.
9. Save the file and test the movie. Open the Bandwidth Profiler and note the file size. Go back to the Flash document and change the field to use Verdana or Trebuchet. Test the movie and note the difference in file size in the Bandwidth Profiler (more about the Bandwidth Profiler later).

Shared ActionScript

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. The following example creates a simple test from a movie called `ScriptLibrary.swf` to a movie called `myMovie.swf`.

1. First you need to create the `ScriptLibrary.swf` movie. Create a new Flash document.
2. Create a new movie clip symbol named `sharedScript`.
3. On frame 1 of the `sharedScript` clip, add the following code:

```
Function demo( ){
    Trace(' 'this is a text, demo' ');
}
```

4. Select the `sharedScript` 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 `sharedScript`.
8. In the URL box, type `codeLib.swf`.
9. Click OK.
10. Save the document as `codeLib.fla`.
11. Export Movie to create `codeLib.swf` from `codeLib.fla`.
12. Close all the files.

We now need to create a file that executes the code imported from the `codeLib.swf`.

1. Save a new Flash document as `NewMovie.fla` in the same folder as `codeLibrary.fla`.
2. Create a layer and call it `sharedCode`.
3. In `File>Open`, choose `codeLib.fla` and open it as Library.
4. Find the instance of the `sharedScript` and drag it on to the stage at frame 1.
5. Name the instance `sharedScript`.
6. Create a new layer on the `NewMovie.fla` timeline and call it `code`.
7. Add a keyframe at frame 2.
8. Add a new frame to the `sharedCode` layer.
9. On frame 2 of the `code` layer, attach this code:

```
Stop( );
sharedScript.demo( )
```

10. Export the movie.

The following text should appear in the output window: “this is a text, demo”. Do not forget, if you change the location of either file you will need to select the symbol in the library, choose linkage and under URL you need to set the new location.

Resolving Conflicts Between Library Assets

If you import or copy a library asset into a movie that already contains a different asset of the same name, you can choose whether to replace the existing item with the new item. This option is available with all the methods for importing or copying library assets, including:

- Copying and pasting an asset from a source movie.
- Dragging an asset from a source movie library.
- Importing an asset.
- Adding a shared library asset from a source movie.
- Using a component from the components panel.

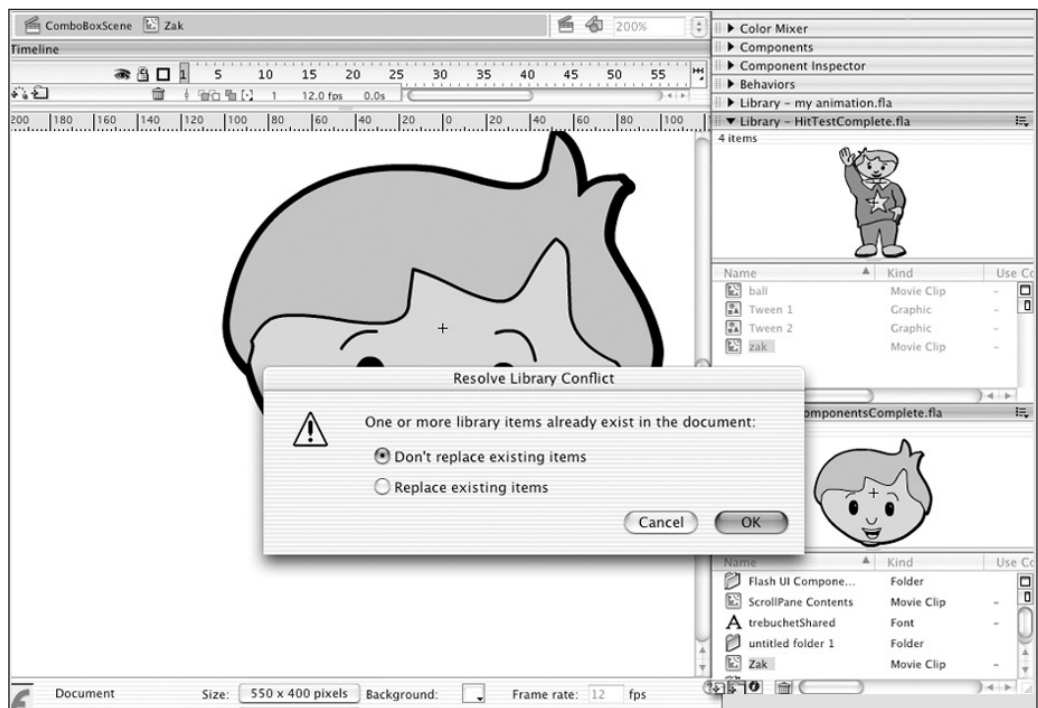


Figure 2.8 *Resolve Library Conflict dialog box*

A conflict exists when you copy an item from a source movie that already exists in the destination movie and the items have different modification dates. At this point you will be prompted to the Resolve Library Items through the dialog. You can organize your assets inside folders in your

movie's library. The dialog box also appears when you paste a symbol or component into your movie's stage and you already have a copy of the symbol or component that has a different modification date from the one you're pasting.

If you choose not to replace the existing items, Flash attempts to use the existing item instead of the conflicting item that you are pasting. The existing items are replaced with the new items of the same name. If you cancel the Import or Copy operation, the operation is canceled for all items. Only identical library item types may be replaced with each other. You cannot replace a sound named Test with a bitmap named Test. In such cases, the new items are added to the library with the word Copy appended to the name. Always save a backup of your FLA file before performing complex paste operations that are resolved by replacing conflicting library items.

Resolving Naming Conflicts Between Library Assets

If you are importing or copying library assets into a movie and you encounter the Resolve Library Conflict dialog box, you need to click Don't Replace Existing Items to preserve the existing assets in the destination movie. Click Replace Existing Items to replace the existing assets and their instances with the new items of the same name.

The shared library is an important tool for centralizing all your code and assets, and can work both locally or across the Internet. Unfortunately this is a little-used tool that can have a major effect on how you work from a workflow to final delivery size to the user on the Internet.

3 Learning about Events and Handlers

Events and Event Handlers

The terms events and event handlers often come up in ActionScript, in fact we have been using both methods in most of the previous exercises. An “event” is an occurrence that triggers an action. An event handler is the thing that can handle the event. Events occur independently of the occurrence of a handler. You need an event to trigger the handler. Events can be grouped into two categories: time-based and user-based. The most common time-based example is that of the playhead entering a frame. Every time the playhead moves to a new frame then an event is generated. All mouse and keyboard activities are described as user events.

The event handlers in Flash are equipped to handle specific events. When you place your actions in the event handler, they get executed when the event occurs. Table 3.1 lists movie clip events and corresponding movie clip event handlers, which some programmers call event methods. Table 3.2 lists the button events and corresponding button event methods.

Table 3.1 *Movie clip events*

Event	Event method
onClipEvent (load)	onLoad
onClipEvent (unload)	onUnload
onClipEvent (enterFrame)	onEnterFrame
onClipEvent (mouseDown)	onMouseDown
onClipEvent (mouseUp)	onMouseUp
onClipEvent (mouseMove)	onMouseMove
onClipEvent (keyDown)	onKeyDown
onClipEvent (keyUp)	onKeyUp
onClipEvent (data)	onData

Table 3.2 *Button events*

Event	Event method
on (press)	onPress
on (release)	onRelease
on (releaseOutside)	onReleaseOutside
on (rollOver)	onRollOver
on (rollOut)	onRollOut
on (dragOver)	onDragOver
on (dragOut)	onDragOut
on (keyPress “...”)	onKeyDown, onKeyUp

Using Actions and Methods to Control Movie Clips

Methods of movie clip objects allow for a variety of functionalities, from timeline control to loading content. Some MovieClip methods perform the same tasks as the actions of the same name; other MovieClip object methods, such as hitTest and swapDepths, do not have corresponding actions.

Actions and methods can have similar behaviors; in fact they can be interchangeable. You can choose to control movie clips by using either one.

Table 3.3 *Actions that target movie clips (see also Chapter 2)*

Actions	Description
loadMovie	<i>X</i> coordinate within the parent movie clip
unloadMovie	<i>Y</i> coordinate within the parent movie clip
loadVariables	Width of object in pixels
setProperty	Height of object in pixels
startDrag	Scale of the object in % <i>X</i> direction
duplicateMovieClip	Scale of the object in % <i>Y</i> direction
_removeMovieClip	Transparency of the object

To use these actions, you must enter a target path for the action's target parameter to indicate the target of the action as we did in the exercise in Chapter 2.

Table 3.4 *MovieClip methods that can control movie clips or loaded levels*

Property	Description
attachMovie	Returns the frame number in which the playhead is located in the timeline
createEmptyMovieClip	Returns the absolute path in slash syntax notation of the movie clip instance on which the MovieClip was dropped
createTextField	Returns the number of frames that have been loaded from a streaming movie
getBounds	Returns a reference to the movie clip or object that contains the current movie clip or object
getBytesLoaded	Returns the target path of the movie clip instance specified in the MovieClip parameter
getBytesTotal	Returns the total number of frames in the movie clip instance specified in the MovieClip parameter
getDepth	Retrieves the URL of the SWF file from which the movie clip was downloaded
globalToLocal	<i>X</i> coordinate of the mouse within the object's space
localToGlobal	<i>Y</i> coordinate of the mouse within the object's space
hitTest	<i>X</i> coordinate of the mouse within the object's space
swapDepths	<i>X</i> coordinate of the mouse within the object's space

To use a method, call it by using the target path of the instance name, a dot, and then the method name and parameters, as in the following statements:

```
_root.gotoAndPlay(10); // Plays the main timeline starting at
frame10
_root.gotoAndPlay('start'); //Plays the main timeline
starting from the label 'start'
```

```
myMovieClip.play( ); // Plays until it comes across another
instruction parentClip.childClip.gotoAndPlay(3); // The
gotoAndPlay method sends the playhead in childClip
    //(which is a child of the instance parentClip) to frame 3 and
    // plays until it finds another instruction
```

startDrag()

Actions that control a timeline have a target parameter that allows you to specify the target path to the instance that you want to control. For example, in the following script the startDrag action targets the customCursor instance and makes it draggable:

```
on(press){
    startDrag('myClip');
}
```

You can use the startDrag() method to allow users to drag a MovieClip object around on stage by using the mouse. You can drag one movie at a time. The startDrag method can have up to five arguments. If no arguments are specified the method moves the MovieClip at the same distance relative to the cursor.

The first argument of the startDrag() method has a Boolean value of either true or false.

MovieClipObject.startDrag(true);

This argument specifies whether or not a MovieClip should lock its center to the cursor. This is useful as it means that whatever the starting position of the MovieClip object, the position after the drag will always match the cursor position. The additional four arguments are positions for a bounding box to lock the cursor and constrain the dragging of a MovieClip object.

You can use the startDrag action or method to make a movie clip draggable while a movie is playing. For example, you can make a draggable movie clip for games, drag-and-drop functions, customizable interfaces, scroll bars and sliders. A movie clip remains draggable until explicitly stopped by stopDrag, or until another movie clip is targeted with startDrag. Only one movie clip can be dragged at a time. To create more complicated drag-and-drop behavior, you can evaluate the “_droptarget” property of the movie clip being dragged. For example, you might examine the “_droptarget” property to see if the movie was dragged to a specific movie clip and then trigger another action.

Try it out with a MovieClip named square by applying the ActionScript to the MovieClip:

```
A Method;
square.startDrag(true, 0, 0, 250, 250);
```


and then change your code to:

```
An Action;  
startDrag(square, true, 0, 0, 250, 250);
```

You will notice that the code works in exactly the same way. Different circumstances will make one more convenient than the other.

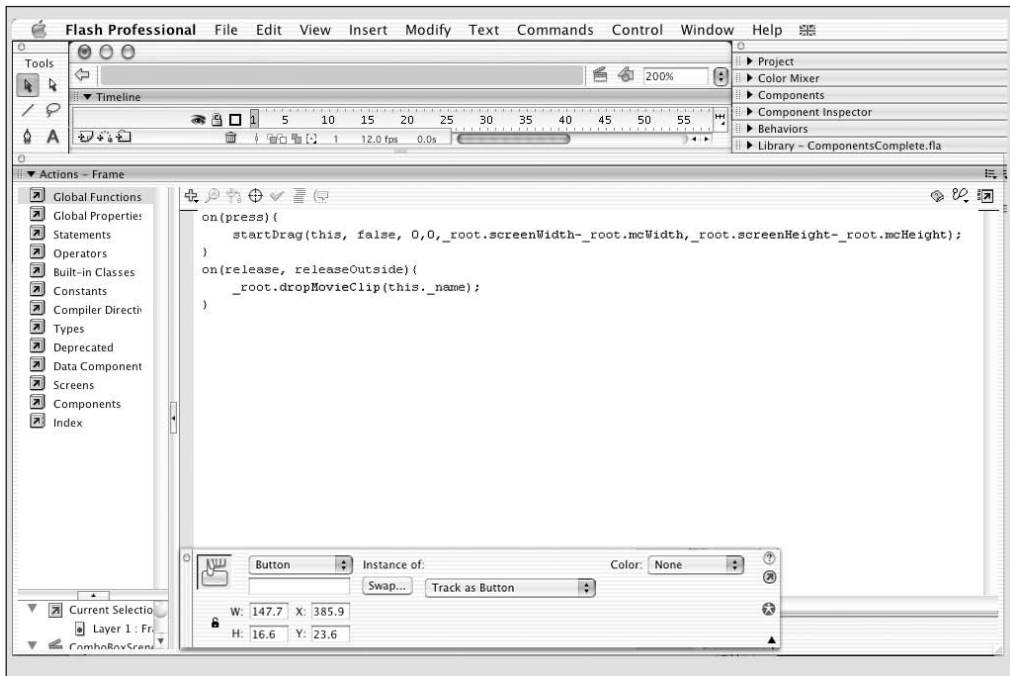


Figure 3.1 *startDrag* statement on a button

The `stopDrag()` works in the same way but requires no arguments, it just releases the object.

```
on (press) {  
    startDrag('myButton', true);  
}  
on (release) {  
    stopDrag();  
}
```

You can find a worked-up example of the `startDrag` action on the website (www.sprite.net/ActionScript/movieClipControl fla). As can be seen from Figure 3.2, we have used a number of

arguments to support the `startDrag` method. These arguments create constraints around where on the stage you can take the draggable movie clip.

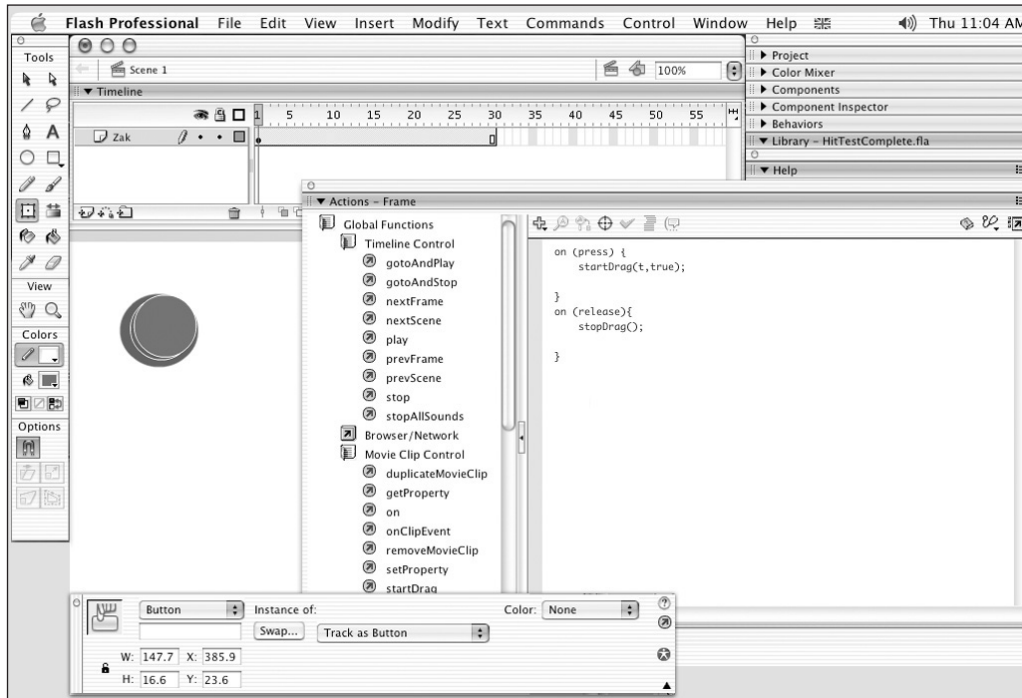


Figure 3.2 *startDrag* statement with constraining arguments from the `movieClipControl.fla`

duplicateMovieClip()

To duplicate or remove movie clip instances as a movie is playing, use `duplicateMovieClip` or `removeMovieClip`. The `duplicateMovieClip` action and method dynamically create a new instance of the movie clip, assign it a new instance name, and give it a depth. A duplicated movie clip always starts at frame 1 even if the original movie clip was on another frame when duplicated, and is always on top of all previously defined movie clips placed on the timeline. To delete a movie clip you created with `duplicateMovieClip`, use `removeMovieClip`. Duplicated movie clips are also removed if the parent movie clip is deleted.

Both of the following statements duplicate the instance `myMovieClip`; try it out on a new clip (`newClip`), and place it at a depth of 10.

```
duplicateMovieClip('myMovieClip', 'newClip', 10);
myMovieClip.duplicateMovieClip('newClip', 10);
```

This method does create something very close to a duplicate of the original movie using the same timeline. The new object has the same values at the original for many of its properties. The exception to this is the “_currentframe” property of the new object; this is always set to 1. You will also need to name your new object and the integer value of the depth of the object.

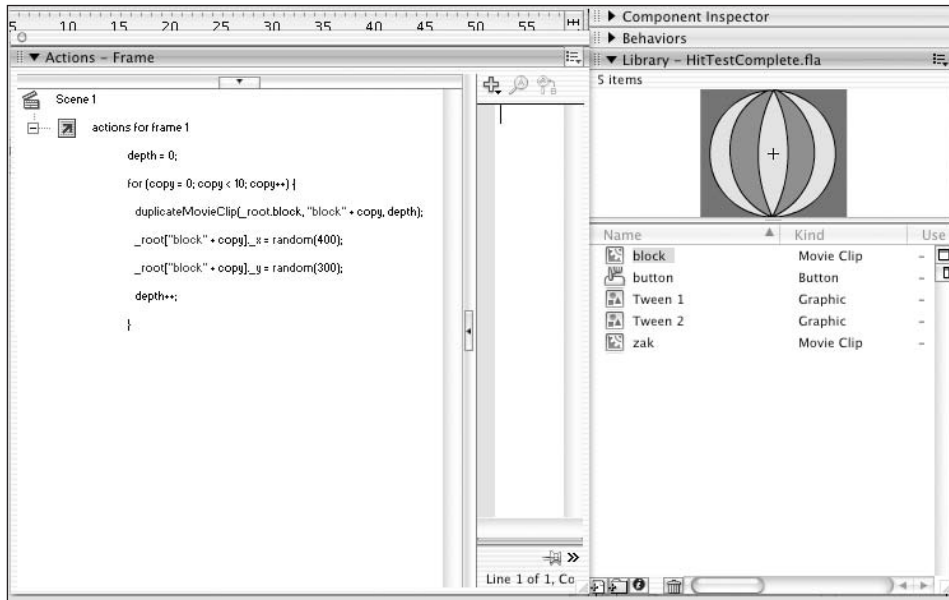


Figure 3.3 Code performing a “for” loop that duplicates 20 MovieClips

This code performs a loop that repeats 20 times. You will learn more about this type of statement in Chapter 4. The first line duplicates an instance of a movie clip that exists on the stage, called block. The duplicateMovieClip function is used to create a copy of an existing movie clip with a new name and at a specific depth on the stage.

In this case, the block movie is duplicated and given the names “block0”, “block1”, “block2” etc. The depth is simply incremented for each copy, since each movie must be at a unique depth.

When you create a new MovieClip this way it is positioned directly on top of its source clip. The duplicated instance whose source clips have been transformed (resized or rotated) will inherit the transformation at duplication time. Any further transformations will not be inherited.

The duplicateMovieClip() offers other advantages over placing clips manually in a movie, such as the ability to control when a clip appears or disappears on the stage.

Multiple Methods on a Single Movie Clip

You can use the “with” action to address a movie clip once, and then execute a series of methods on that clip. The “with” action works on all ActionScript objects (for example, Array, Color and Sound), not just movie clips. The “with” action takes an object as an argument. The object you specify is added to the end of the current target path. All actions nested inside a “with” action are carried out inside the new target path, or scope. For example, in the following script, the “with” action is passed the object `Zak.face` to change the properties of `face`:

```
with (Zak.face){
    _rotation = 45;
    _alpha = 20;
    _xscale = 150;
    _yscale = 150;
}
```

It is as if the statements inside the “with” action were called from the timeline of the hole instance. The above code is equivalent to the following:

```
Zak.face._rotation = 45;
Zak.face._alpha = 20;
Zak.face._xscale = 150;
Zak.face._yscale = 150;
```

The above code is also equivalent to the following:

```
with (zak){
    face._rotation = 20;
    face._alpha = 20;
    face._xscale = 150;
    face._yscale = 150;
}
```

With, tellTarget and Dot Syntax

The “with” statement is a shorthand way to reference properties of an object without having to retype the object’s name. It takes the form:

```
with(object){
    statement
}
```

The “with” function is used to change the scope of a block of code. It is a way to execute a block of ActionScript in another movie clip from outside that movie clip. If a movie had an instance of a movie clip called “car”, to play the “car” movie from the root you could use any of the following:

```
car.play();
```

or

```
with(car) {  
    play();  
}
```

or

```
tellTarget('car') {  
    play();  
}
```

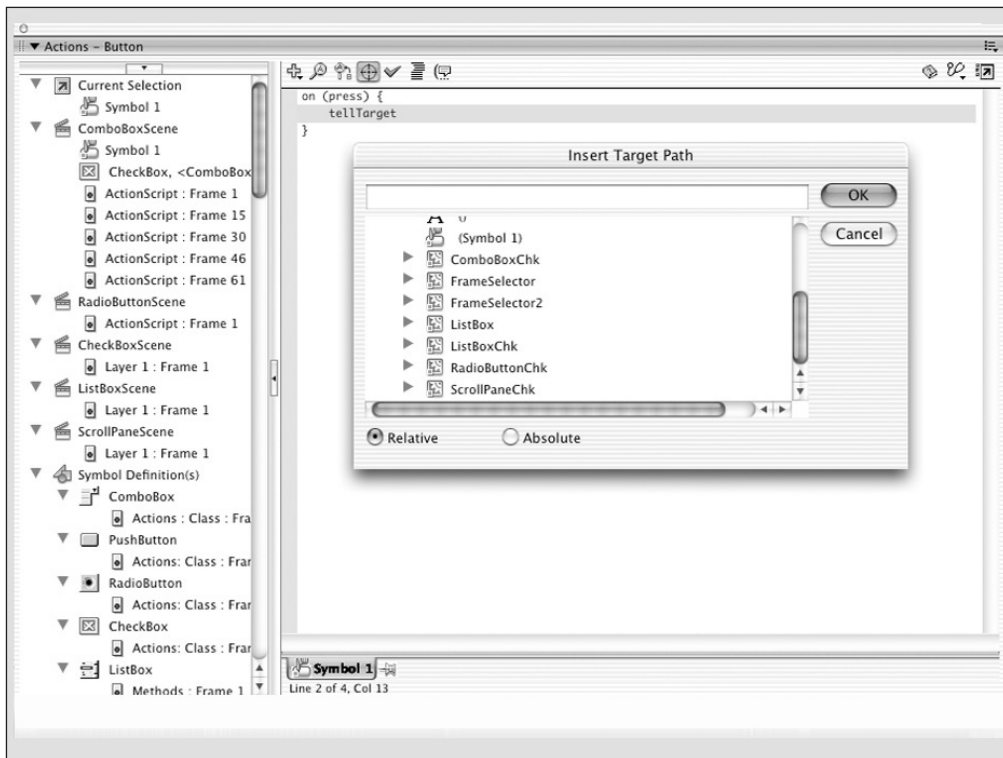


Figure 3.4 The *tellTarget* method is still heavily used

The `tellTarget` method has been deprecated and replaced with the “with” function; however, you will still encounter `tellTarget` in older code. The advantage of using “with” over the usual dot syntax is that it can remove the need for you having to repeatedly write the object’s name or the path to the object. For example:

```
_root.Car.Wheel.steering = _root.Car.Wheel.steering
                        + _root.Car.Wheel.steerAmount;
_root.Car.Wheel.turn( );
_root.Car.Wheel.gotoAndPlay(_root.Car.Wheel.nextFrame);
```

can be neatly replaced with:

```
with (_root.Car.Wheel) {
    steering = steering + steerAmount;
    turn( );
    gotoAndPlay(nextFrame);
}
```

Loading Sounds and Images Dynamically

The new `loadSound(URL, isStreaming)` method allows you to load MP3 files into Flash movies at runtime. The `attachSound()` method works with sound files that have been imported into the Flash MX document and exported with the Flash movie. You can use the `loadSound` method of the `Sound` object to load MP3 sound files into a Flash movie as it plays. For `loadSound(URL, isStreaming)` files, Flash supports only the MP3 sound file type.

To load a sound dynamically:

1. Create a new file.
2. Choose **Window>Development Panels>Actions**.
3. In the Actions toolbox, click the **Actions** category, click **Variables**, and double-click the **set variable** action to add it to the Script pane.
4. In the **Variable** parameter box, enter an instance name for the new object, `mySound`.
5. With the insertion point in the **Value** parameter box, from the Actions toolbox, click the **Objects** category, click **Movie**, click **Sound**, and double-click **new Sound** to add it to the Script pane. Select the **Expression** box.
6. The code should look like this:

```
mySound = new Sound( );
```

7. In the Actions toolbox, click the **Objects** category, click **Movie**, **Sound** and **Methods**, and double-click the `loadSound` method to add it to the Script pane.
8. In the **Object** parameter box, enter the instance name of the movie clip into which the sound will load, `mySound`.

9. In the Parameters box, enter the URL at which the sound is located. Put a comma (,) after the URL. Enclose the URL in quotation marks – for example, “http://www.sprite.net/Sounds/sound14.mp3”.
10. After the comma in the Parameters box, enter the value false for the isStreaming parameter to indicate that the sound is an event, as follows:

```
this.loadSound('http://www.sprite.net/Sounds/sound14.mp3',
false)
```

11. In the Actions toolbox, click the Objects category, click Movie, Sound and Methods, and double-click the start method to add it to the Script pane.
12. In the Object parameter box, enter the instance name of the sound to start, mySound.

The code should look like this:

```
mySound = new Sound( );
mySound.loadSound('http://www.sprite.net/Sounds/
sound14.mp3', true);
mySound.start( );
```

To load an image or an SWF into a level in the Flash Player, you must use the loadMovieNum method or action. To load an image into a movie clip target in the Flash Player, you must use the loadMovie action or method. The loaded image replaces all the contents of the target movie clip. When you load an image, the upper left corner of the image is placed on the registration point of the movie clip. Because this registration point is often the center of the movie clip, the loaded image may not appear centered. Also, when you load an image to a root timeline, the upper left corner of the image is placed on the upper left corner of the Stage. The loaded image inherits rotation and scaling from the movie clip, but the original content of the movie clip is removed. Flash supports only the standard JPEG image file type, not progressive JPEG files.

To load an SWF dynamically:

1. Create a new file.
2. Choose Window>Development Panels>Actions.
3. In the Actions toolbox, click the Objects category, click Movie, MovieClip and Methods, and double-click the loadMovie method to add it to the Script pane.
4. In the Object parameter box, enter the instance name of the movie clip into which the image will load – in this example, myMC. If the movie clip is not a child of the same parent as the timeline that calls the action, you must use a target path. You can use an absolute or relative path.
5. In the Parameters box, enter the URL at which the image is located. Enter a comma (,) after the URL.
6. After the comma in the Parameters box, enter the HTTP method “GET” or “POST” (in quotation marks), or leave the parameter blank.

The following code loads an image into a movie clip on the timeline where the movie clip that calls the action is located:

```
myMC.loadMovie('http://www.sprite.net/understanding/
image1.jpg')
```

Changing Movie Clip Position and Appearance

To change the properties of a movie clip as it plays, write a statement that assigns a value to a property or use the `setProperty` action. For example, the following code sets the rotation of instance `mc` to 45:

```
MovieClip._rotation = 45;
```

This is equivalent to the following code, which uses the `setProperty` action:

```
setProperty('MovieClip', _rotation, 45);
```

In Flash MX both the `MovieClip` and the `Button` objects can be controlled by setting the properties. The properties and their results are set out in Table 3.5.

Some properties, called read-only properties, have values that you can read but not set. The read-only properties are listed in Table 3.6.

Table 3.5 *Appearance properties*

Property	Description
<code>_x</code>	<i>X</i> coordinate within the parent movie clip
<code>_y</code>	<i>Y</i> coordinate within the parent movie clip
<code>_width</code>	Width of object in pixels
<code>_height</code>	Height of object in pixels
<code>_xscale</code>	Scale of the object in % <i>X</i> direction
<code>_yscale</code>	Scale of the object in % <i>Y</i> direction
<code>_alpha</code>	Transparency of the object
<code>_visible</code>	Can make the object either visible or invisible
<code>_rotation</code>	Rotation of object in degrees

Table 3.6 *Read-only properties*

Property	Description
<code>_currentframe</code>	Returns the frame number in which the playhead is located in the timeline
<code>_droptarget</code>	Returns the absolute path in slash syntax notation of the movie clip instance on which the <code>MovieClip</code> was dropped
<code>_framesloaded</code>	Returns the number of frames that have been loaded from a streaming movie
<code>_parent</code>	Returns a reference to the movie clip or object that contains the current movie clip or object
<code>_target</code>	Returns the target path of the movie clip instance specified in the <code>MovieClip</code> parameter
<code>_totalframes</code>	Returns the total number of frames in the movie clip instance specified in the <code>MovieClip</code> parameter
<code>_url</code>	Retrieves the URL of the SWF file from which the movie clip was downloaded
<code>_xmouse</code>	<i>X</i> coordinate of the mouse within the object's space
<code>_ymouse</code>	<i>Y</i> coordinate of the mouse within the object's space

You can write statements to set any property that is not read-only. The following statement sets the `_alpha` property of the movie clip instance `eye`, which is a child of the `face` instance:

```
face.eye._alpha = 50;
```

You can write statements that get the value of a movie clip property. The following statement gets the value of the `_xmouse` property on the current level's timeline and sets the `_x` property of the `customCursor` instance to that value:

```
onClipEvent(enterFrame){  
    customCursor._x = _root._xmouse;  
}
```

This is equivalent to the following code, which uses the `getProperty` function:

```
onClipEvent(enterFrame){  
    customCursor._x = getProperty(_root, _xmouse);  
}
```

The `_x`, `_y`, `_rotation`, `_xscale`, `_yscale`, `_height`, `_width`, `_alpha` and `_visible` properties are affected by transformations on the movie clip's parent, and transform the movie clip and any of the clip's children. The `_focusrect`, `_highquality`, `_quality` and `_soundbuftime` properties are global; they belong only to the level 0 main timeline.

Dynamically Adding a Movie Clip or Sound to the Stage

To retrieve a copy of a movie clip or sound from the library and play it as part of your movie, you use the `attachMovie` method of the `MovieClip` object or the `attachSound` method of the `Sound` object. The `attachMovie` method loads a movie clip as a child of the clip that loads it and plays it as the movie runs. The `attachSound` method attaches a sound to an instance of the `Sound` object.

To use ActionScript to attach a movie clip or sound from the library, you must assign a unique linkage identifier to the movie clip or sound. You can assign this name in the Linkage Properties dialog box.

When a movie plays, Flash loads all movie clips and sounds that are added with `attachMovie` or `attachSound` before the first frame of the movie. This can create a delay before the first frame plays. When you assign a linkage identifier to an element, you can also specify whether this content should be added before the first frame. If it isn't added in the first frame, you must include an instance of it in some other frame of the movie; if you don't, the element will not be exported to the SWF file.

To name a movie clip:

1. From within your file, choose Window>Library to open the Library panel.
2. Select a movie clip in the Library panel.
3. In the Library panel, choose Linkage from the Library panel options menu.
4. The Linkage Properties dialog box appears.
5. For Linkage, select Export for ActionScript.
6. For Identifier, enter an ID for the movie clip.
7. If you don't want the movie clip or sound to load before the first frame, deselect the Export in First Frame option.
8. Click OK and save your file.

To attach a movie clip to another movie clip:

1. With the Actions panel open, select a frame in the timeline.
2. In the Actions toolbox (at the left of the Actions panel), click the Objects category, the Movie category and the MovieClip category, and double-click the attachMovie method. For the object parameter, enter the instance name of a movie clip on the Stage.

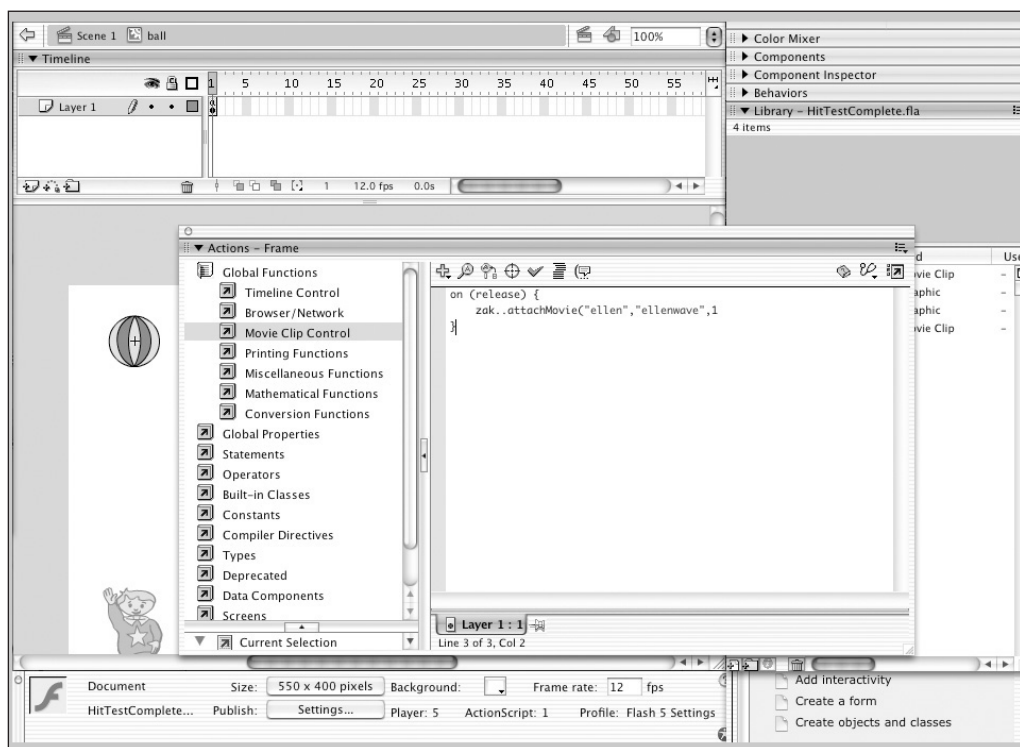


Figure 3.5 Movie with attachMovie method referenced by a button

3. Enter values for the following parameters:
 For `idName`, specify the identifier you entered in the Linkage Properties dialog box.
 For `newName`, enter an instance name for the attached clip so that you will be able to target it.
 For `depth`, enter the level at which the duplicate movie will be attached to the movie clip. Each attached movie has its own stacking order, with level 0 as the level of the originating movie. Attached movie clips are always on top of the original movie clip, as in Figure 3.5, `Zak_mc.attachMovie("ellen", "ellenWave", 1)`.

Dynamically Creating an Empty Movie Clip

The `createEmptyMovieClip` method allows you to create an empty `MovieClip` on the stage while a movie plays, using the `createEmptyMovieClip` method of the `MovieClip` object. This method creates a movie clip as a child of the clip that calls the method. The registration point for a newly created empty movie clip is the upper left corner. For example:

```
_root.createEmptyMovieClip('emptyMc', 1);
```

This creates a new `MovieClip` object on the timeline called `emptyMc` with a depth of 1.

Although the `createEmptyMovieClip` method behaves similarly to `attachMovie`, you don't need to provide a linkage identifier because you aren't adding a symbol from the library.

To create an empty movie clip:

1. Select a button or frame to assign the action.
2. Choose **Window>Actions**.
3. In the Actions toolbox, click the **Objects** category, the **Movie** category, the **MovieClip** category and the **Methods** category, and double-click `createEmptyMovieClip`.
4. For the object parameter, enter the instance name of a movie clip on the stage, or click the **Insert Target Path** button to browse to an instance.
5. Enter values for the following parameters:
 For `instanceName`, specify an identifier.
 For `depth`, enter the level at which the duplicate movie will be attached to the movie clip. Each created movie has its own stacking order, with level 0 as the level of the originating movie. Newly created movie clips are always on top of the original movie clip.

Here is an example:

```
myMovieClip.createEmptyMovieClip('newMC', 10);
```

Empty movie clips are useful if you need to load movies into a changing location on the stage. The empty movie clip is used as the destination for the new movie clip.

Scriptable Masks

You can use a movie clip as a mask to create a hole for contents of another movie clip. The mask movie clip plays all the frames in its timeline and can be controlled by ActionScript. You can make the mask movie clip draggable, animate it along a motion guide, use separate shapes within a single mask, or resize a mask dynamically. If you decide you no longer wish a masked object to be masked, you can call the `setMask()` method and pass it the null value.

```
Mask_mc.setMask(null);
```

You cannot use a mask to mask another mask. You cannot set the `_alpha` property of a mask movie clip. Only fills are used in a movie clip that is used as a mask; strokes are ignored.

To create a mask:

1. Choose a movie clip to be masked.
2. In the Property Inspector, enter an instance name for the movie clip “pic”.
3. Create a movie clip to be a mask. Give it an instance name in the Property Inspector, such as mask.
4. Select frame 1 in the timeline.
5. Choose Window>Actions.
6. In the Actions toolbox, click the Objects>the Movie>the MovieClip, and the Methods category, and double-click `setMask`.
7. In the parameters area, enter the instance name of the mask movie clip.

The code should look like this:

```
pic.setMask(mask);
```

A Small Application – A Preloader

A preloader is typically the first scene in a movie which displays a loading screen until the whole movie has loaded. Once completely loaded the preloader will start the movie, this stops the movie from loading assets as it plays, causing it to pause. This exercise looks at a range of new methods but it is also a recap on how these methods and actions come together.

To add a preloader:

1. Create a new scene in your movie and set it to be the first scene played by moving it to the top of the scene list.
2. Next create a new layer named “ActionScript” and create keyframes on frames 1 and 2.
3. Create a new layer named “Preloader” and place on it any assets you want to appear on the preloader. Make sure the assets appear for the first two frames.
4. Stretching a movie clip as the movie loads creates the progress bar. Create a new layer called “Progress Bar”.

5. Download preloader.fla (from www.sprite.net/understanding) and drag the “Progress Bar” asset onto the “Progress Bar” layer in to your movie. Give the movie clip the instance name “progbar”. Add the following code to the first frame action:

```
// Work out how much of the movie has been loaded
pcent_loaded = (getBytesLoaded( ) / getBytesTotal( )) * 100;

progbar._width = pcent_loaded;

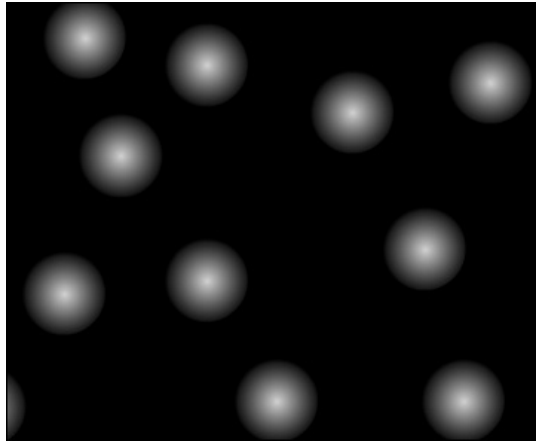
// If all bytes are loaded move onto the next scene
if (getBytesLoaded( ) >= getBytesTotal( )) {
    nextScene( );
    play( );
}
```

6. Finally, add the following code to the second frame action:

```
gotoAndPlay(1);
```

The code on the first frame action does all the work. It uses the `getBytesLoaded()` function to determine how many bytes of the movie have been loaded, and the `getBytesTotal()` method to determine how many bytes the whole movie is. Using these two values it calculates the percentage loaded and resizes the progress bar. It then compares the two byte values; if they are equal it goes to the next scene and starts playing, since all bytes have now been loaded. Otherwise it will simply loop by playing the `gotoAndPlay(1)` on frame 2.

Great, your first application that you can use again and again!



ACTIONSCRIPT FUNDAMENTALS

02

This Page Intentionally Left Blank

4 Variables, “If”, “For” and “While” Loops

Variables

In order to retrieve and manage information we need to store data as variables; this is the primary information storage container of ActionScript. A variable is a container that stores a piece of data. The container itself is always the same, but the contents can change. Variables can be used to store information as the movie plays and can be used for many situations, such as to change the behavior of the movie based on the user interaction. I always liken a variable to a bank account, but instead of holding money it holds data. When you create a new variable it is like setting up a new account, so your new data is stored in another location different from the previous data. All you have to do is give the variable a different name so you can remember what data resides with what variable. Data in the variable account can change; you can add to it or take away from it. Using the variable (data account) to store information becomes a fixed reference to access changing content.

Variables can hold any type of data: number, string, Boolean, object or movie clip. Typical types of information you can store in a variable include a URL, a user’s name, the result of a mathematical operation, the number of times an event occurred, or whether a button has been clicked. Each movie and movie clip instance has its own set of variables, with each variable having its own value independent of variables in other movies or movie clips.

Each variable must be assigned a name, which is not a reserved keyword (such as function, if, for, etc.) or a Boolean value (true/false). You must be aware that variable names must be composed of letters, numbers, dollar signs (\$) and underscores. You cannot have spaces, hyphens or punctuation marks. You must start with a letter, underscore or a dollar sign. The variable name should not exceed 255 characters. Finally, it is a convention in ActionScript to start the variable name as lower case. These are legal names:

```
mystring = 'http://www.sprite.net';  
mynumber = 123;  
q1_question = 'What is a variable?';  
q1_answer = 'A contains that stores information';  
result = mynumber + 123;  
done = false;    // A Boolean value
```

The single equals (=) operator is used to assign a value to a variable, while the double equals (==) is used to compare two values to determine whether they are equal. These two operators can be easily confused so be careful.

It is good form to append suffixes to your variable name to indicate the type of information stored in the variable name. Flash will activate code hinting in the ActionScript panel.

```
var name_str;
```

In order to take advantage of the code hinting, variables must be named in a specific format. The default format is to append a suffix that indicates the variable type. Table 4.1 lists the supported suffixes.

Table 4.1 *Supported suffixes*

Data type represented	Suffix	example
String	_str	myString_str
Array	_array	myArray_array
MovieClip	_mc	myMovieClip_mc
TextField	_txt	myTextField_txt
Date	_date	myDate_date
Sound	_sound	mySound_sound
XML	_xml	myXML_xml
Color	_color	myColor_color
Button	_btn	myButton_btn
Video	_video	MyVideo_video

An alternative to Flash's variable naming conventions is known as Hungarian Notation. This was developed by Charles Simonyi at Microsoft. You can read more about this on <http://ootips.org/hungarian-notation.html>. This book will teach the Flash way of doing things, so it is easy to follow up material using the Flash online help.

Creating a variable is called a declaration. When a variable is first declared, it is empty and is said to be undefined. To declare a new statement you can name the variable anything you want. The word “var” tells the computer that you are declaring a variable:

```
var name;
var age;
var y;
```

These variables currently have no value.

If you are going to declare variables, it is good practice to declare them at the beginning of every movie; this is usually the first keyframe that comes after a movie's preloader. Remember to comment your code; later in the book we will discuss the importance of good commenting.

You will notice that the following variables do not use the “var” label and I have used labels like “mynumber”. This is because the label can be anything just as long as you keep it the same for the container that holds that particular data.

These are my variables:

```
mystring = 'http://www.sprite.net';
mynumber = 123;
q1_question = 'What is a variable?';
q1_answer = 'A contains that stores information';
result = mynumber + 123;
done = false;    // A Boolean value
```

Changing Variable Values

Creating variables is only useful if you can assign values to them. To retrieve a variable's values, take the variable's name and assign a value after the = sign:

```
var name = 10; //declare name and assign it a numeric value.
```

```
name = 'Bertie'; //assign name a text value
```

What we did here is change the datatype from numeric to text data by simply assigning it a value of the desired type. ActionScript is unusual in allowing you to change the data type; most programming languages will not allow you to do this.

A variable's “scope” refers to the area in which the variable is known and can be referenced. Variables in ActionScript can be either global or local. A global variable is shared among all time-lines, i.e. a global variable in one movie can be accessed by another movie using the dot syntax.

```
var myvar = _root.movieA.movieAA.anothervariable;
```

The `_root` is a reference to the main timeline of a movie. We can address variables on the main movie timeline using the `_root`. The following are a few examples:

```
_root.firstName //access the variable firstName on the main
timeline
_root.familyName //access the variable familyName on the main
timeline
```

We can combine the reference to root with the reference to the movie clip instance.

```
_root.myMovie.firstName
```

What you see here is the variable `firstName` in `myMovie` that resides on the main movie timeline.

A local variable is only available within its own block of code (between the curly braces). A local variable is defined using the “var” keyword; examples of both global and local variables are shown below. Functions will be explained in detail in the next chapter.

```

myglobal = 'Hello, I'm a global variable' ;
testlocal( ) ;

function testlocal( ) {
    // mylocal is not available outside the testlocal function
    var mylocal = 'Hello, I'm a local variable' ;
    trace(mylocal) ;
    trace(myglobal) ;
}
trace(mylocal) ; // this will not print 'Hello, I'm a...'
                // as mylocal is not defined outside the
                // testlocal function.
trace(myglobal) ; // this will print, since myglobal
                // is defined globally.

```

Loading External Variables

In most instances you will create your variables inside Flash, but you will find yourself needing to load variables from an external text file or web page. Loading external variables actually creates new variables at runtime. You can do this using four variable loading techniques.

LoadVars()

The LoadVars object lets you send all the variables in an object to a specified URL and load all the variables at a specified URL into an object. Because this is an object, the convention in both Flash and other programming languages is to start the first letter in the name with an upper case letter. You must create a new instance of the LoadVars object to call its methods. This instance is a container to hold the loaded data. Follow the steps to LoadVars object.

1. Create a new movie.
2. Choose a frame to assign the action.
3. Open Window>Actions.
4. In the Actions toolbox, click the Actions category, click Variables and double-click the set variable action to add it to the Script pane. In the Variable parameter box, enter an instance name for the new object – for example, myVars.
5. With the insertion point in the Value parameter box, from the Actions toolbox, click the Objects category, then click Client/Server, click LoadVars and double-click new LoadVars. The code should look like this:

```
myVars = new LoadVars( ) ;
```

6. In the Actions toolbox, click the Objects category, click Client/Server, LoadVars and Methods, and double-click the load method to add it to the Script pane.
7. In the Object parameter box, enter the instance name of the LoadVars object into which the data will load – in this example, myVars.
8. In the Parameters box, enter the URL from which to download data.

The URL must be enclosed in quotation marks – for example, “http://www.myserver.com/data.txt”. The finished code should look like this:

```
myLoadVars = new LoadVars( );
myVars.load('http://www.myserver.com/data.txt');
```

LoadVariables() reads data from an external file, such as a text file or text generated by a CGI script, Active Server Pages (ASP) or PHP, or Perl script, and sets the values for variables in a Flash Player level or a target movie clip. This action can also be used to update variables in the active movie with new values. This example loads information from a text file into text fields into the myMovie movie clip on the main timeline. The variable names of the text fields must match the variable names in the data.txt file.

```
on(release) {
    loadVariables('data.txt', '_root.myMovie');
}
```

Fscommand()

This action lets you extend your movie by using the capabilities of the host. You can pass an fscommand action to a JavaScript function in an HTML page that opens a new browser window with specific properties. To control a movie in the Flash Player from web browser scripting languages such as JavaScript, VBScript and Microsoft JScript, you can use Flash Player methods – functions that send messages from a host environment to the Flash movie. For example, you could have a link in an HTML page that sends your Flash movie to a specific frame.

The next set of steps illustrate how to open a message box from a Flash movie in HTML through JavaScript.

1. In the HTML page that embeds the Flash movie, add the following JavaScript code:

```
<SCRIPT>
function theMovie_DoFSCommand(command, args) {
    if (command == 'messagebox') {
        alert(args);
    }
}
</SCRIPT>
```

If you publish your movie using the Flash with FSCommand template in the HTML Publish Settings dialog box, this code is inserted automatically. The movie’s NAME and ID attributes will be the file name. For example, for the file myMovie.fla, the attributes would be set to myMovie.

2. In the Flash document, add the fscommand action to a button, as shown in this example:

```
fscommand('messagebox', 'This is a message box invoked from
within Flash.')
```

You can also use expressions for the `fscommand` action and parameters, as in this example:

```
fscommand('messagebox', 'Hello, ''+name+', welcome to our  
Web site!')
```

3. Choose File>Publish Preview>HTML to test the document.

Operators

An operator is a symbol or keyword that is used for manipulating data. Most people are familiar with things like + (addition) and – (subtraction). You will know the symbol * as multiplication but you are probably used to seeing it as ×. In programming languages generally, you will find that the × is replaced by *. So the following example means 2 multiplied by 4:

```
2 * 4;
```

This multiplication can be applied to every possible data type. The data or values are described as operands and they can be in any kind of multiplication. Here is an example

```
Total = highScore * players
```

The operands can be in any kind of expression and can also in themselves be made up of expressions – for example:

```
averageHighScore = (playerOneScore + playerTwoScore) /  
numberOfPlayers
```

What we are being told above is to take the `playerOneScore` and add to it `playerTwoScore` and then divide this by the number of players. What is important is that you name your variables and make them descriptive of the data.

Operator Precedence

Some operators take precedence over others; this is very useful when two or more operators are used in the same statement. ActionScript follows a precise hierarchy to determine which operators to execute first. For example, multiplication is always performed before addition; however, items in parentheses take precedence over multiplication. So, without parentheses, ActionScript performs the multiplication in the following example first:

```
total = 4 + 5 * 3; // results in 19, because 4 + 15
```

This expression is evaluated as `4 + (5 * 3)` because the * has higher precedence than the + operator. But when parentheses surround the addition operation, ActionScript performs the addition first:

```
total = (2 + 4) * 3;
```

The result is 18.

Operator List

Table 4.2 lists all of the ActionScript operators and their associativity, from highest to lowest precedence.

Table 4.2 *ActionScript operators and their associativity*

Operator	Description	Associativity
Highest precedence		
+	Unary plus	Right to left
−	Unary minus	Right to left
~	Bitwise one’s complement	Right to left
!	Logical NOT	Right to left
not	Logical NOT (Flash 4 style)	Right to left
++	Post-increment	Left to right
--	Post-decrement	Left to right
()	Function call	Left to right
[]	Array element	Left to right
.	Structure member	Left to right
++	Pre-increment	Right to left
--	Pre-decrement	Right to left
new	Allocate object	Right to left
delete	Deallocate object	Right to left
typeof	Type of object	Right to left
void	Returns undefined value	Right to left
*	Multiply	Left to right
/	Divide	Left to right
%	Modulo	Left to right
+	Add	Left to right
add	String concatenation (formerly &)	Left to right
−	Subtract	Left to right
<<	Bitwise left shift	Left to right
>>	Bitwise right shift	Left to right
>>>	Bitwise right shift (unsigned)	Left to right
<	Less than	Left to right
<=	Less than or equal to	Left to right
>	Greater than	Left to right
>=	Greater than or equal to	Left to right
lt	Less than (string version)	Left to right
le	Less than or equal to (string version)	Left to right
gt	Greater than (string version)	Left to right
ge	Greater than or equal to (string version)	Left to right
==	Equal	Left to right
!=	Not equal	Left to right
eq	Equal (string version)	Left to right
ne	Not equal (string version)	Left to right
&	Bitwise AND	Left to right
^	Bitwise XOR	Left to right
	Bitwise OR	Left to right

(continued)

Table 4.2 *(continued)*

Operator	Description	Associativity
Highest precedence		
&&	Logical AND	Left to right
and	Logical AND (Flash 4)	Left to right
	Logical OR	Left to right
or	Logical OR (Flash 4)	Left to right
?:	Conditional	Right to left
=	Assignment	Right to left
*=, /=, %=, +=, -=, &=, =, ^=, <=<=, >>=, >>>=	Compound assignment	Right to left
,	Multiple evaluation	Left to right
Lowest precedence		

Operator Associativity

When two or more operators share the same precedence, their associativity determines the order in which they are performed. Associativity can be either left to right or right to left. For example, the multiplication operator has an associativity of left to right; therefore, the following two statements are equivalent:

```
total = 2 * 3 * 4;
total = (2 * 3) * 4;
```

The “=” operator is right to left, so in the following expression:

```
a = b = c = d
```

assign d to c, then assign c to b, and then assign b to a, so it looks like this:

```
a = (b = (c = d))
```

Incremental Operators

The most common usage of the increment operator is `i++`. An alternative way to describe this would be:

```
i = i+1
```

You can use the increment operator before or after an operand. In the following example, age is incremented first and then tested against the number 26:

```
if (++age >= 26)
```

This is like saying if age = age add 1 for as long as it is more than 26.

In the following example, age is incremented after the test is performed:

```
if (age++ >= 30)
```

Table 4.3 lists the ActionScript numeric operators:

Table 4.3 *ActionScript numeric operators*

Operator	Operation performed
+	Addition
*	Multiplication
/	Division
%	Modulo (remainder of division)
-	Subtraction
++	Increment
--	Decrement

Comparison Operators

Comparison operators compare the values of expressions and return a Boolean value (true or false). These operators are most commonly used in loops and in conditional statements. Comparison operators are intended for comparing strings and numbers. When two operands of a comparison operator are numbers, the comparison is performed mathematically: $3 < 10$ is true and $3 > 10$ is false. When the two operands of a comparison operator are strings, the comparison is performed to the character code points. This means that upper and lower case letters that have different points are not considered equal. The following values all evaluate to false because they are not equal:

```
''Ben Salter'' == ''ben salter''
''Robert Vacher'' == ''RobertVacher''
```

Using the `!=` we can make the above evaluate to true:

```
''Ben Salter'' != ''ben salter''
''Robert Vacher'' != ''RobertVacher''
```

In the following example, if the variable score is 100, a certain movie loads; otherwise, a different movie loads:

```
if (score>100){
    loadMovieNum(''winner.swf'', 5);
} else {
    loadMovieNum(''loser.swf'', 5);
}
```


Table 4.4 lists the ActionScript comparison operators.

Table 4.4 *ActionScript comparison operators*

Operator	Operation performed
<	Less than
>	Greater than
<=	Less than or equal
>=	Greater than or equal

String Operators

The `+` operator has a special effect when it operates on strings: it concatenates the two string operands. For example, the following statement adds “Ben,” to “Salter”:

```
''Ben, '' + ''Salter''
```

The result is “Ben, Salter”. If only one of the `+` operator’s operands is a string, Flash converts the other operand to a string.

The comparison operators `>`, `>=`, `<` and `<=` also have a special effect when operating on strings. These operators compare two strings to determine which is first in alphabetical order. The comparison operators only compare strings if both operands are strings. If only one of the operands is a string, ActionScript converts both operands to numbers and performs a numeric comparison. Try this out on a button in a new movie:

```
on (release) {
    test = ''z'' < ''a'';
    if (test != true) {
        trace(''winner'');
    } else {
        trace(''loser'');
    }
}
```

So what you are doing above is testing the numeric value of “z” against the value of “a”. As the numeric value of z is greater than the value of a, the answer is not true. The inequality operator `!=` is really confirming that. Read on to learn how equality or inequality operators work.

Equality Operators

You can use the equality (`==`) operator to determine whether the values or identities of two operands are equal. You will frequently use equality operations to form Boolean expressions within conditional statements or to assign a Boolean value to a variable. This comparison returns

a Boolean (true or false) value. If the operands are strings, numbers or Boolean values, they are compared by value. If the operands are objects or arrays, they are compared by reference.

It is a common mistake to use the assignment operator to check for equality. For example, the following code compares *x* to 2:

```
if ( x == 2 )
```

In that same example, the expression *x* = 2 is incorrect because it doesn’t compare the operands, it assigns the value of 2 to the variable *x*.

The strict equality (===) operator is like the equality operator, with one important difference: the strict equality operator does not perform type conversion. If the two operands are of different types, the strict equality operator returns false. The strict inequality (!==) operator returns the inversion of the strict equality operator. Table 4.5 lists the ActionScript equality operators.

Table 4.5 *ActionScript equality operators*

Operator	Operation performed
==	Equality
===	Strict equality
!=	Inequality
!==	Strict inequality

Logical Operators

Logical operators compare Boolean (true and false) values and return a third Boolean value. The logical operator OR is most commonly used to initiate some action when at least one of the two conditions is met. For example, if both operands evaluate to true, the logical AND operator (&&) returns true. If one or both of the operands evaluate to true, the logical OR operator (||) returns true. Logical operators are often used in conjunction with comparison operators to determine the condition of an if action. In the following script, if both expressions are true, the “if” action will execute:

```
if ( i>9 && _framesloaded>80 ){
    play( );
}
```

A logical AND operation is a conditional test expression. Here is an example of this:

```
X = 99;
Y = 150;
```

```

if(x>80 && Y>120){ //only when both of two conditions are met is
this statement true
    trace('both x and y are greater than 80');
}

```

Table 4.6 lists the ActionScript logical operators.

Table 4.6 *ActionScript logical operators*

Operator	Operation performed
&&	Logical AND
	Logical OR
!	Logical NOT

Bitwise Operators

Bitwise operators internally manipulate floating-point numbers to change them into 32-bit integers. Bitwise operators are only important if you are building a large application where memory and speed need to be optimized. The exact operation performed depends on the operator, but all bitwise operations evaluate each binary digit (bit) of the 32-bit integer individually to compute a new value. Table 4.7 lists the ActionScript bitwise operators.

Table 4.7 *ActionScript bitwise operators*

Operator	Operation performed
&	Bitwise AND
	Bitwise OR
^	Bitwise XOR
~	Bitwise NOT
<<	Shift left
>>	Shift right
>>>	Shift right zero fill

Assignment Operators

You can use the assignment (=) operator to assign a value to a variable, as in the following:

```
name = 'Fred';
```

name is a variable in which we want to place the value Fred. Fred represents the value that we want to store.

You can also use the assignment operator to assign multiple variables in the same expression. In the following statement, the value of c is assigned to the variables a and b:

```
a = b = c;
```

You can also use compound assignment operators to combine operations. Compound operators perform on both operands and then assign that new value to the first operand.

Table 4.8 lists the ActionScript assignment operators.

Table 4.8 *ActionScript assignment operators*

Operator	Operation performed
=	Assignment
+=	Addition and assignment
-=	Subtraction and assignment
*=	Multiplication and assignment
%=	Modulo and assignment
/=	Division and assignment
<<=	Bitwise shift left and assignment
>>=	Bitwise shift right and assignment
>>>=	Shift right zero fill and assignment
^=	Bitwise XOR and assignment
=	Bitwise OR and assignment
&=	Bitwise AND and assignment

Dot and Array Access Operators

The dot operator is our primary means of referring to object properties and nested movie clips. The dot operator has the exact same purpose as the [] operator; it allows the programmer to set and retrieve object–property values. You can use the dot operator (.) and the array access operator ([]) to access built-in or customized ActionScript object properties, including those of a movie clip. The dot operator uses the name of an object on its left side and the name of a property or variable on its right side. The property or variable name can’t be a string or a variable that evaluates to a string; it must be an identifier. The following examples use the dot operator:

```
Object.property
```

The dot operator and the array access operator perform the same role, but the dot operator takes an identifier as its property, whereas the array access operator evaluates its contents to a name and then accesses the value of that named property. For example, the following expressions access the same variable speed in the movie clip car:

```
car.speed;
car['speed'];
```

You can use the array access operator to dynamically set and retrieve instance names and variables. For example, in the following code, the expression inside the [] operator is evaluated and the result of the evaluation is used as the name of the variable to be retrieved from the movie clip name:

```
name['mc' + i]
```

You can also use the “eval” function, as shown here:

```
eval(''mc'' + i)
```

The array access operator can also be used on the left side of an assignment statement. This allows you to dynamically set instance, variable and object names, as in the following example:

```
name[index] = ''Alex'';
```

Conditionals

A conditional is a type of statement that performs an action only when a specified condition is met. We can use conditionals to create and control situations that have more than one potential outcome. 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 met, then an alternative block is executed. In essence what is happening is all conditions either allow or suppress the execution of a code block based on a condition.

The “If” Statement

“If” statements are used to check whether a certain condition is true or false. If the condition is true, ActionScript executes the statement block that follows. If the condition doesn’t exist, ActionScript skips to the next statement outside the block of code. For example, the following code checks to see if the variable “test” is true.

```
test = false;
if (test == true) {
    trace(''Yep, test is true'');
}
```

An “if” statement can also be used to check more than one condition and provide a block of code that is executed if all the conditions are false. This is done using the “if...else...” statement. For example, open actionscriptdemo2.fla and add the following code to the first frame action:

```
test = ''fifteen'';
if (test == ''one'') {
    stop();
} else if (test == ''fifteen'') {
    gotoAndStop(15);
} else if (test == ''thirty'') {
    gotoAndStop(30);
} else {
    gotoAndStop(45);
}
```

This code compares the value of “test” to determine which frame to move the movie to. If test is not “one”, “fifteen” or “thirty” then it will jump to frame 45 in the final “else” block.

The condition is generally constructed using the following comparison operators:

<code>A < B</code>	- A is less than B
<code>A > B</code>	- A is greater than B
<code>A <= B</code>	- A is less than, or equal to B
<code>A >= B</code>	- A is greater than, or equal to B
<code>A == B</code>	- A is equal to B
<code>A != B</code>	- A is not equal to B

The “While” Statement

No language would be complete without some form of iteration (repeated execution of a block of program code). Flash offers several of these methods, the simplest of which is the “while” loop.

This simply executes a block of code repeatedly *while* a specific expression is true. For example,

```
while (done == false) {
    ... some game functions ...
    if (endofgame == true) {
        done = true; // this tells the while loop to stop
    }
}
// “while” loop jumps to here when it has finished.
```

The expression is evaluated before the code is run, if it is true the code is executed and when finished the expression is re-evaluated. This continues until the expression is false, whereby it jumps to the end of the block. You can also check the “while” condition at the end of the loop by using a “do...while” loop. This means that the loop is always executed at least once before the expression is evaluated – for example:

```
do {
    ... some game functions ...
    if (endofgame == true) {
        done = true;
    }
} while (done == false);
```

However, generally you will find that a “for” loop proves to be more useful.

The “For” Statement

A “for” loop can be used to repeat a block of code *for* a specific number of times. It requires three expressions, one to start the loop, another to test whether it should continue and a third to increment the loop. For example,

```
for(count = 0; count <= 10; count++) {
    trace(count);
}
```

In this case, the starting condition (count = 0) is evaluated first, before the loop is started, to set the count variable to 0. Next, the continue condition is evaluated (count <= 10) and the loop is repeated if it is true; in this case, if “count” is less than or equal to 10. The body of the expression is then executed, which prints the number to the output window. Finally, the loop increment expression is evaluated (count++), which adds one to the count variable. The continue condition is then re-evaluated and the cycle then repeats until the count reaches 11.

To demonstrate this, open `actiondemo3.fla` (www.sprite.net/understanding) and add the following code to the first frame action:

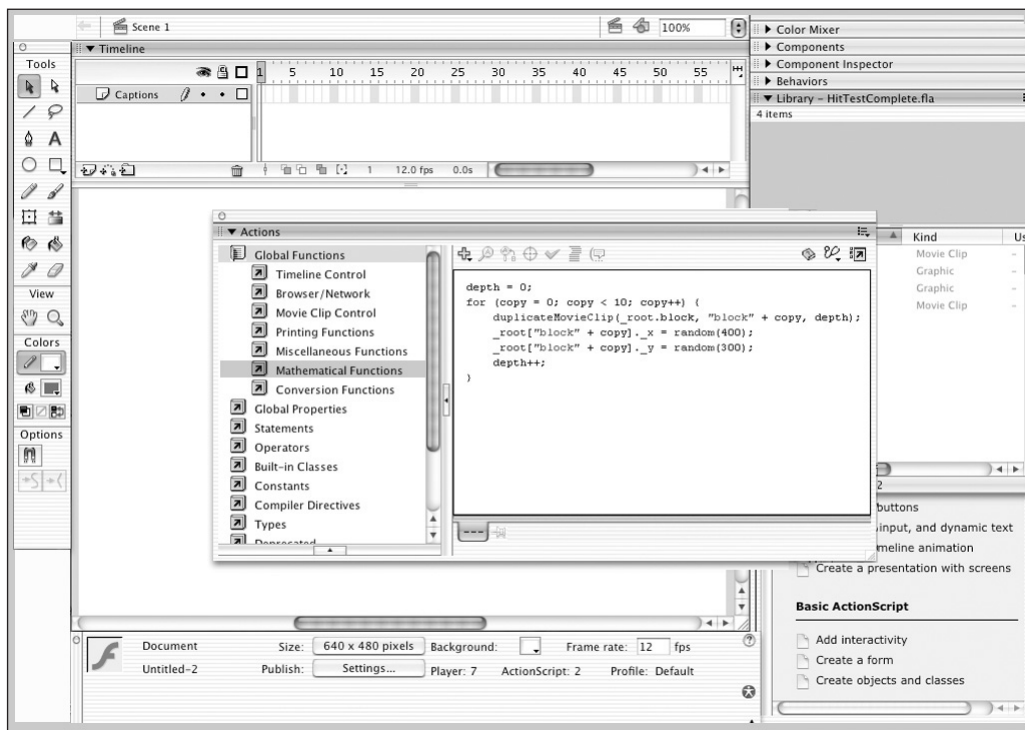


Figure 4.1 Code is applied to the first frame of the movie

```

var newmoviedepth = 0;
for (copy = 0; copy < 20; copy++) {
    duplicateMovieClip(_root.block, 'block'+copy,
newmoviedepth);
    newmoviedepth++;
    _root['block'+copy]._x = random(400);
    _root['block'+copy]._y = random(300);
}

```

This code performs a “for” loop that repeats 20 times. In the “for” block, the first line duplicates an instance of a movie clip that exists on the stage, called block. The duplicateMovieClip function is used to create a copy of an existing movie clip with a new name and at a specific depth on the stage.

In this case, the block movie is duplicated and given the name “block0”, “block1”, “block2” etc. The depth is simply incremented for each copy since each movie must be at a unique depth.

The next two lines make use of the random() function to place the newly created movie copy at a random position on the stage. The random(num) function simply generates and returns a random integer between 0 and the specified number, num. An alternative is to use the Math.random() function, which generates a random decimal number between 0 and 1.

The “for” loop repeats this process to create 20 copies of the block movie. Experiment by changing the values in the “for” loop and random functions, and if you’re feeling adventurous try modifying the code to duplicate “blockblue” as well.

The Conditional Operator

The conditional operator (?:) is a convenient and simple way of expressing what can be normally four or five lines of code. Here is an example:

```
condition ? value1 : value2;
```

What we are saying here is that if the condition is true, value1 is returned. Otherwise value2 is returned. This conditional operator is a quicker way to write the following conditional statement:

```

if(condition){
    value1
}
else{
    value2
}

```

You can use the conditional operator to control a program. Here is an example:

```
action == 'go' ? play() : stop();
```


What we are saying here is if action is “go”, play the movie, otherwise stop.

The Switch Statement

The switch statement, introduced in Flash MX, allows you to execute several possible code blocks, based on a value of a single test expression. The general form of the switch statement is:

```
switch(testExpression){
  case expression1:
    substatement1
    break;
  case expression2:
    substatement2
    break;
  default:
    substatement3
    break;
}
```

The expression is evaluated to see if the expression is equal to the test. If it is equal it executes all the remaining statements in the switch block. Here is an example that has a bug in it – see if you can spot the problem:

```
num = 10;
switch(num){
  case 20:
    trace('20');
  case 10
    trace('10');
  case 5:
    trace('5');
}
```

The output for this will be:

```
10
5
```

The first time I encountered this I thought that the result would be 10 and that it would not have continued to execute the following block. To do this properly you have to add a break statement. Here it is again:

```
num = 10;
switch(num){
  case 20:
```

```

        trace('20');
        break;
    case 10:
        trace('10');
        break;
    case 5:
        trace('5');
    }

```

The output for this will be:

```
10
```

A catch-all equivalent will be to add a default statement at the end of the last break:

```

num = 10;
switch(num){
    case 20:
        trace('20');
        break;
    case 10:
        trace('10');
        break;
    case 5:
        trace('5');
    default:
        trace('no case was met')
}

```

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. All scripting and programming languages use data. Understanding how this data is stored is an important part of learning about the internal tools within ActionScript. In the next chapter we will look at functions and arrays.

5 Functions and Arrays

This chapter looks at our last two corners of the fundamentals of ActionScript, arrays and functions. Both concepts once understood, will revolutionize the way you build applications.

Functions

Functions are a way of packaging a block of code that performs a particular task so that it can be used by a shorthand call, eliminating the need to retype the code. Functions encapsulate code that is used repeatedly. They are invoked by “calling” the function’s name. In addition to reducing the amount of typing a programmer must do, functions generally make code more readable by replacing large blocks of complicated instructions with a simple, easy-to-digest function name. A well-written function can be used by other movies. ActionScript has many functions built into it, like `trace()`. These are special types of functions; another built-in function that you will often use is the `gotoAndStop()` function. A built-in function is a reusable bit of code that comes built into ActionScript.

A function consists of a name and a block of statements with variables referenced in the block which may include variable declarations. Here is an example:

```
function name(a, b, c, ...) {  
    statement  
}
```

The function keyword starts the declaration that this is a function. The name is an identifier for the function; it must be a name unique to that block of code. Then we have the parameters also referred to as “arguments”, which are enclosed within parentheses; a comma separates each parameter. Then we need to provide the code that is executed when this function is called. Once a function is defined, it can be called from any timeline, including the timeline of a loaded movie. Like variables, each function is defined with a name, and can also define a set of variables that it expects to receive. For example, the following function moves a movie clip “ball”, on the root movie, to a defined coordinate. This should be defined in a frame action on the root movie. Try it:

```
function moveball(x, y) {  
    _root.ball._x = x;    // set the x co-ordinate of the ball  
    _root.ball._y = y;    // set the y co-ordinate of the ball  
}
```

This function is called by a button on the root movie, using the following code:

```
on (release) {
    // move the ball to x=100,y=200
    moveball(100,200);
}
```

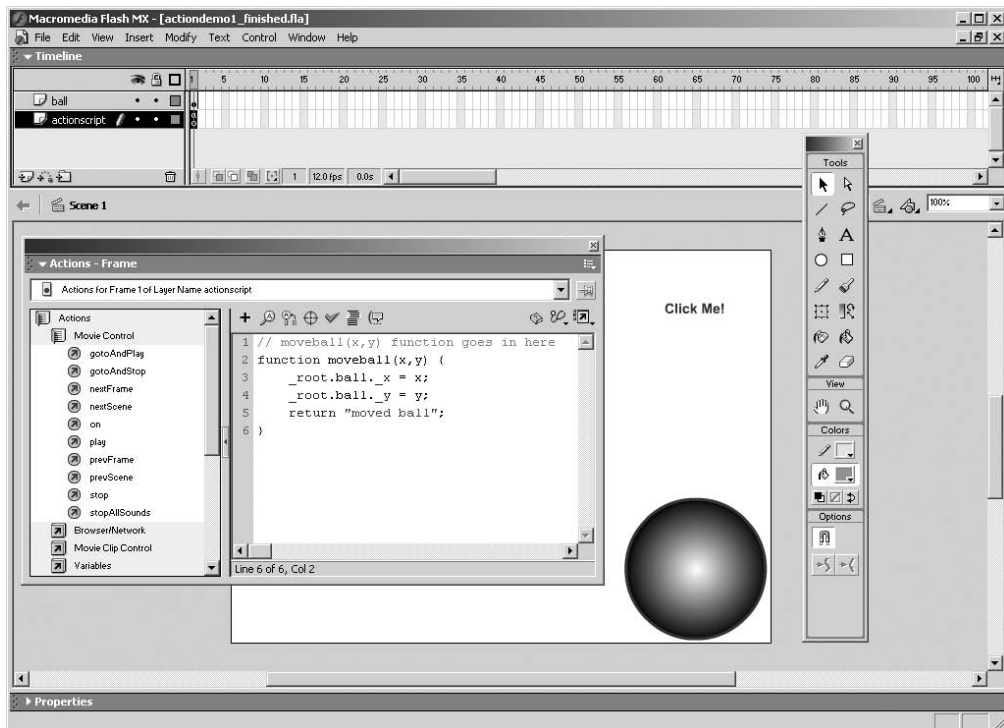


Figure 5.1 Code for “moveball” movie

If a function requires no parameters the parentheses are left empty. The following example is of a function that has no parameters with a function call `sayHello()`:

```
function sayHello() {
    trace('hello!');
}
```

This is called by the declaration or “invocation” `sayHello()`; the term “declaration” refers to the definition of the function. Everything from the “function” keyword to the final “`}`” is the function’s declaration. “`var temp;`” is a variable declaration. The function is “called” or “invoked” or “executed” by the “invocation”, “`sayHello()`”.

The semicolon at the end of the function call is there because it makes up a complete statement and all statements should end in a semicolon. This function has no parameters.

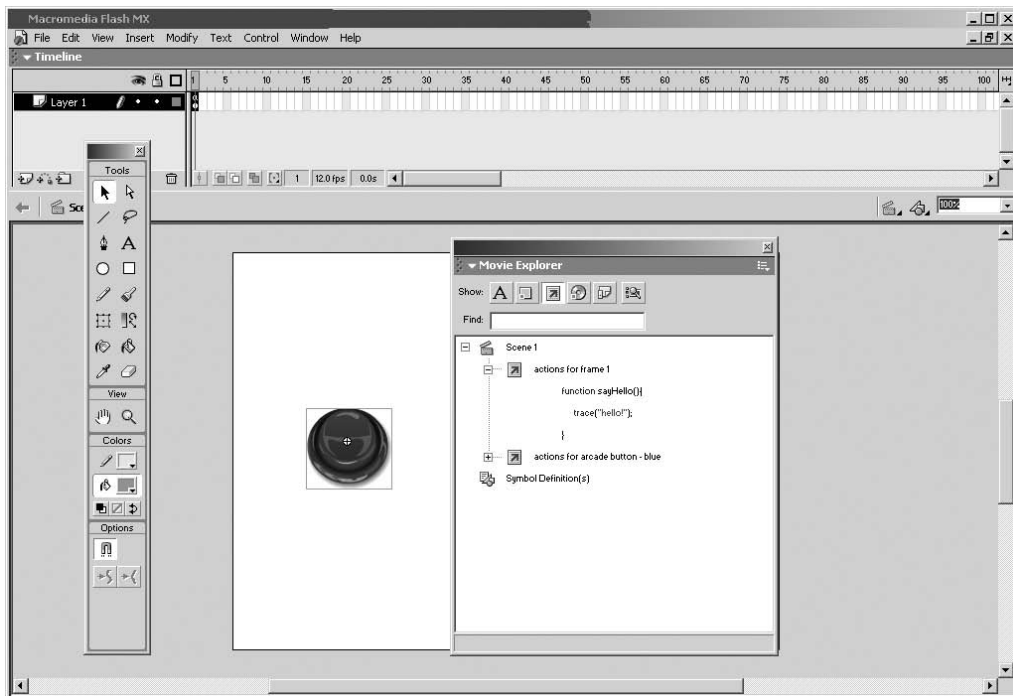


Figure 5.2 *Movie Explorer with function call inside it*

Functions with Parameters

The function's parameters are assigned values when the function is executed. They act as placeholders for data values. Functions can also return values, which can be useful for testing values – for example, modify moveball to return a string value when it has finished:

```
function moveball(x, y) {
    _root.ball._x = x;
    _root.ball._y = y;
    return "moved ball"; // return the string "moved ball"
}
```

The return statement can be used to terminate a function and optionally to return a result. When a return statement is encountered during a function execution, it skips any remaining statements in the function. The return tells the interpreter to return control to the script containing the function invocation. If no return statement is available, ActionScript acts as if the last line of the function body contains a return statement.

Now modify the button code to print the returned value:

```
on (release) {
    // move the ball to x=100, y=200, putting the return value in
```

```

    // "result", and displaying it in the output window.
    result = moveball(100,200);
    trace(result);
}

```

You should now see “moved ball” printed in the output window when you click the button. Functions are a useful way of breaking up large amounts of code, as well as defining code that can be reused.

Returning Values from Functions

The return as in the moveball(x,y) function is also used to send a value back to the script that invokes the function, like this:

```
return "moved ball"; // return the string "moved ball"
```

The result of the expression becomes the result of the function invocation. Let's try it again:

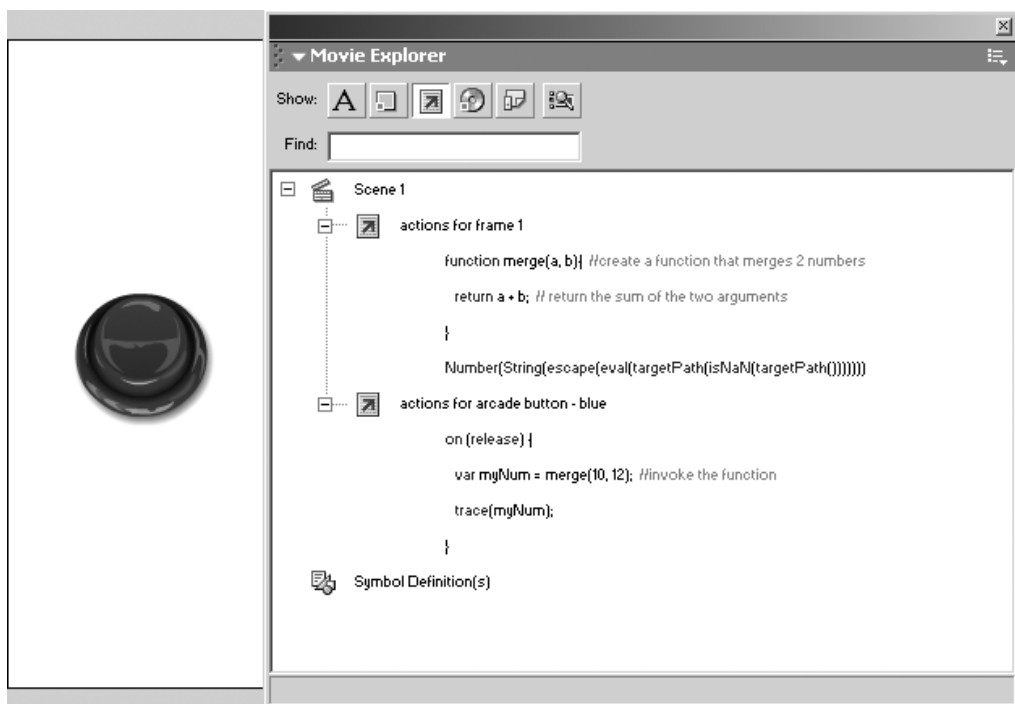


Figure 5.3 Movie Explorer with function *merge(a, b)* call inside it

The merge() function calculates and returns the sum of two numbers. It does not perform an action as in the example sayHello(), which displayed a message. We can make use of a function's return value by assigning it to a variable, as in this example:

```
var myNum = merge(10, 12);
```

The result of a function call is an ordinary expression and so can be used as part of additional expressions, like this:

```
var myNum = merge(10, 12) + ' ' + 'turned up for lunch';
```

If a return statement doesn't include an expression to be returned the return statement is missed out; a function will probably return the value undefined. The following examples will not evaluate anything meaningful, because the return statement is missing:

```
function merge(a, b){  
    var result = a + b; // returned but not calculated  
}
```

```
function merge(a, b){  
    var result = a + b;  
    return; // no return value to return  
}
```

```
function merge(a, b){  
    var result = a + b;  
    return result; // returns the desired value.
```

Types of Function

Although functions do not need to be standardized, you will find it very helpful if you are going to reuse functions to standardize them. You will notice that functions fall into one of four categories.

Subroutines

This is when a function does not return a value but does something, like moves a MovieClip across the screen or any process. When you use functions in this way they are known as statements. When you use subroutines you call them to invoke a statement, like we did with the ball MovieClip earlier on in this chapter.

Data

This is when a function operates like a variable and returns a value.

Functions within functions

Functions within functions can also be defined on any timeline, i.e. in MovieClip assets in the library. For example, open the ball asset and add the following function to the first frame action of the movie:

```
function changealpha(newalpha) {  
    // set the alpha channel to the new value  
    this._alpha = newalpha;  
}
```

And modify the button code to read:

```
on (release) {
    result = moveball(100,200);
    trace(result);
    _root.ball.changealpha(50); // make the ball
    semi-transparent
}
```

Value passing

Parameters provide input on which functions can operate. Functions in ActionScript pass values in two ways: by reference and by value. The difference has to do with the data; types such as numbers, strings and Boolean are passed by value. What this means is that the true or literal value is passed to the function and there is no link back to the original variable. Here is an example of this:

```
Function myfunction (a){
    a++
}
myVal = 5
myFunction(myVal);
trace(myVal);
```

In this example the value of “myVal” was passed to “myFunction();” and was assigned a local variable within the function. Even though the value is increased by one within the function “myVal” retains its value. Local variables are valuable tools for organizing code and making it easier to understand. When a function uses local variables, it can hide its variables from all other scripts in the movie; local variables are scoped to the function and are destroyed when the function exits. All parameters passed to a function are also treated as local variables.

The values used as arguments can be any legitimate expressions from the ActionScript syntax. For example:

```
function message(today){
    trace(' 'my message to you on' ' + today);
}
```

To Invoke message() we call a statement like this:

```
on (release) {
    today = new date( );
    message(today);
}
```


or like this:

```
on (release) {
    message(290403);
}
```

You will notice that “my message to you on”, today’s date and 290403 all belong to different data types. This is interesting and unusual amongst programming languages; ActionScript allows you to pass any data of any type to a function as long as it understands what to do with the past values. To pass more than one argument to a function you need to separate each argument with a comma. If you recall, our `moveball(x,y)` function accepted two arguments, `x` and `y`. The arguments used when invoking a function can be any valid data type, including literals and identifiers. The parameters in the function declaration in the body of the function are always identifiers. That is, they are always names of variables that are local to the function.

Function Scope

When you declare a function in the standard way, it is scoped from within the timeline and can be called by its name or its target path. If the function was within another function, it is accessible only within that parent function. When you assign an anonymous function a name, you are actually assigning a reference to the anonymous function to a variable. Here is an example:

```
var myFunction = function Scope( ) {}
```

Rather than a variable containing a value, it contains a reference to the function. So you can assign a scope to the variable and therefore assign a reference to an anonymous function to a global variable. This makes it available throughout the movie without having to use a target path. For example:

```
_global.setTimeout = function(func, delay){
    var id = setInterval( );
    func( );
}
```

This is really an example of the use of “_global” since all functions are declared in this same way.

This creates a global function “setTimeout” that invokes a function “func” after a certain delay.

Built-in Functions

Flash comes with its own predefined functions. We have already seen some of these at work manipulating data. For example:

```
gotoAndPlay(15);
```

The `gotoAndPlay` is a good example of this. A playhead of a movie clip calls the function `gotoAndPlay()` and sets a frame number as a parameter. Custom functions typically do something

that a built-in function does not offer and, like any custom functions, you need to specify a name, value or argument. By remembering certain actions as functions you find them easier to use.

The Actions Toolbox has a folder labeled Functions. In this folder you find functions, but there is also another folder called Actions. The only difference between the two is that the functions return values and actions do not return values.

Centralizing Code for Functions

Whilst I have never been a great supporter of centralizing all code within a complex movie, Flash is very unstructured in this sense. Macromedia even encourages authors to tuck code inside movie clips, buttons, etc. The main disadvantage of doing this is that it makes it very difficult for more than one programmer to work on any given FLA. In a large-scale production environment, centralizing code makes it possible for a programmer to open an FLA created by another programmer, and quickly find all the important code in a standardized, centralized location. In some cases it dramatically increases the value of our code by making it more reusable. I am a supporter of externalizing generalized functions so they can be reused in other applications of the future. If we have a series of buttons in a movie clip all with the same purpose, instead of multiplying our code by the number of buttons we can put the function on the main timeline or in an external file (.as) and call it from each button. This has a number of benefits; in a complex function it saves time and reduces the potential for errors. Code modularity and centralization makes it easier to reuse all or part of the code in other projects. A well-written function should be able to be plugged into many different applications as it should be independent of the rest of the program you have created. This is why it is important to give your functions names that describe the task they perform and keep their function coding to those tasks. After all, you should build this so it is general, in that it can be used with lots of different applications, but the function needs to be specific to a purpose. An example of this type of function is one that has a credit card clearing function that can be invoked from anywhere, but its purpose is to interface with the bank's server to clear credit cards. The main drawback of ".as" files (or any externalization) is that multiple files must be kept together in order to build the SWF. Keeping everything in the FLA provides a very convenient encapsulation.

You should avoid using global variables or variables that have been defined outside the functions. If you need to assign a value to a variable that will be used outside the scope of the function, you should use a return statement instead. If you should find the return statement limiting, you should break up your function into smaller functions.

What is an Array?

An array is a convenient way to group related information. Typically arrays are used to store list of things, like names, coordinates, etc. An array is simply an ordered data structure that can encompass multiple data values. These structures are made up of elements of values with indexes or keys

that correspond to those values. It is likened to an index with a number of values. We use arrays to do everything from storing values to generating pull-down menus. In the most basic form, an array is just like a list of items.

Array Size

At any point during its life, an array can contain a specific number of elements. This number of elements is called the array length. We can insert and delete elements from the beginning, end or even middle of an array.

The Array Object

Array objects are used to store an indexed set of objects or variables. For example, an array could contain a high score table, list of player names or the status of the game. Each element in the array has an index that is used to retrieve that element. The element can be a reference to any Flash object; typically these will be one of the standard data types (String, Number, Boolean), although they can also be a reference to any other Flash object, even another Array object. The Array object lets you access and manipulate arrays. An array is an object whose properties are identified by a number representing their position in the array. All arrays are zero-based, which means that the first element in the array is [0], the second element is [1], and so on. By storing Array objects in an array, it allows you to build multi-dimensional arrays, but more on this later.

We speak of an array as containing many values, but you should always remember that an array is a single datum of information, like the string “peter” is a single string containing multiple characters. An array can be assigned to a variable or used as part of an expression:

```
myChildren = [ 'Ellen, Alex, Zak, Malchay' ];
// an array with my childrens names in
display(myChildren); // pass the array to a function
```

Each item in an array is called an array element, and adopts a unique numeric position by which we can refer to it.

As well as the standard way of setting and retrieving elements of the array using their index, the Array object also provides other useful methods for manipulating them. The current size of the array is always stored in the length property of the Array object; this is useful for determining the end of the array. For example,

myArray.length = 5

“first”	“second”	“third”	“fourth”	“fifth”
---------	----------	---------	----------	---------

myArray[0] myArray[1] myArray[2] myArray[3] myArray[4]

The first way you can create an array is to create an array literal by enclosing a comma-delimited list of values in square brackets, as in the `myChildren` example above, which created a string of values. Here are a few more examples:

```
[1, 2, 3, 4];
[red, blue, green];
[1 + 2, 3 * 2, 4];
```

Here are the same examples using an `Array()` constructor:

```
new Array (1, 2, 3, 4)
new Array (red, blue, green)
new Array (1 + 2, 3 * 2, 4)
```

The simplest way to create an `Array` object with a constructor is to call the `Array` constructor with the `new` operator and no arguments (we will discuss constructors in a later chapter; for this chapter just assume it is something that allows you to build a new object array):

```
myArray = new Array ( );
```

You can then manually load the array in this way:

```
myArray[0] = "first element";
myArray[1] = "second element";
myArray[2] = "third element";
```

Retrieving elements from an array:

```
first = myArray[0];
third = myArray[2];
arrayLength = myArray.length;
```

Array index example:

myArray.length = 5				
"first"	"second"	"third"	"fourth"	"fifth"
myArray[0]	myArray[1]	myArray[2]	myArray[3]	myArray[4]

The “for” loop is ideal for iterating through an array. This example demonstrates how to print each value of an array to the Output window:

```
myArray = new Array ( );
myArray[0] = "first element";
myArray[1] = "second element";
myArray[2] = "third element";
```

```

for(count = 0; count < myArray.length; count++) {
    trace('Item ' + count + ' is ' + myArray[count]);
}

```

The “for” loop, as in the previous example, loops through the array elements. The counter for the loop is “count”, which is smaller than the total length of the array, as indicated in this fragment of code:

```
count < myArray.length;
```

Length of an Array

The length property holds the integer value of the number of elements in an array. As the number of elements in an array changes, so does the value of its property. Here is an example exploring this:

```

MyArray = new Array(3);
Trace(myArray.length);
MyArray[5] = 'element 6';
Trace(myArray.length);
MyArray[24] = 'element 25';
Trace(myArray.length);

```

Figure 5.4 illustrates code with Output window result for the code used.

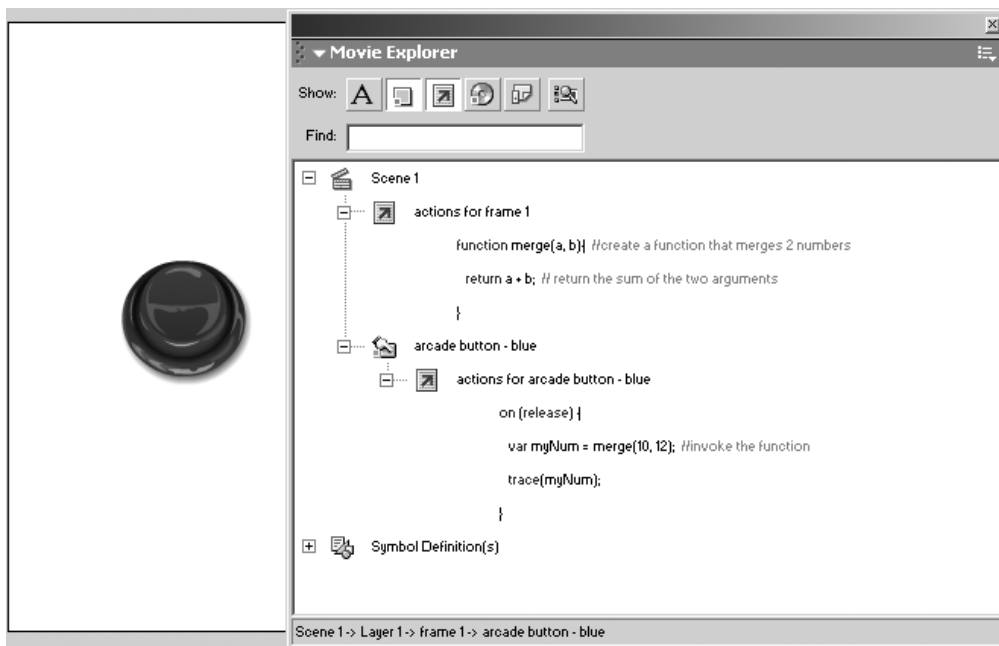


Figure 5.4 Code with Output window result

The length of an array is always equal to one more than the index of the last element. You will also notice that ActionScript arrays do not need to be contiguous or fully defined, but the length is always greater than the last element's index. The length property is often used in “for” statements. This is done by creating a “for” statement that will iterate over the entire length of an array, no matter what the length:

```
for(count = 0; count < myArray.length; count++) {
    trace('Item ' + count + ' is ' + myArray[count]);
}
```

Assignment Operator

You can write values to the index specified by using the assignment operator:

```
firstArray = new Array('a','b','c');
secondArray = new Array();
secondArray[0] = firstArray[0];
secondArray[1] = firstArray[1];
secondArray[2] = firstArray[2];
```

When you use the assignment operator (=) to assign any object, including arrays to a variable, the reference is assigned and only the reference to the original object is assigned, so it is not a duplicate.

Methods of Array Object

We can use the built-in array methods to handle more complex array operations. The `concat()` and `join()` methods of Array objects enable you to create new Array objects that use some or all of the values from the original object. You can then manipulate the new arrays without having to affect the originals.

concat()

The method `concat()` creates a new array and concatenates the elements specified in the parameters, if any, with the elements in “myArray”. If the “value” parameters specify an array, the elements of that array are concatenated, rather than the array itself. The array “myArray” is left unchanged. Here is an example. The following code concatenates three arrays into a fourth array:

```
num1=[1,3,5];
num2=[2,4,6];
num3=[7,8,9];
nums=num1.concat(num2,num3)
trace(nums); // creates array [1,3,5,2,4,6,7,8,9]
```

This method takes as many parameters as elements you want to append. The parameters can be of any data type.

join()

The join method does not work on associative arrays. If you want to emulate the join() for an associative array you must use the “for” loop. The following example creates an array with three elements. It then joins the array three times – using the default separator, a comma and a space, and a plus sign – and displays them in the Output window:

```
a = new Array('Zak', 'Alex', 'Ellen')
trace(a.join()); // returns Zak, Alex, Ellen
trace(a.join(' - ')); // returns Zak -- Alex - Ellen
trace(a.join(' + ')); // returns Zak + Alex + Ellen
```

slice()

The slice method returns a new array that contains a slice of the elements from the original array. It extracts a slice or a substring of the array and returns it as a new array without modifying the original array. The returned array includes the start element and all elements up to, but not including, the end element. “start” is a number specifying the index of the starting point for the slice. If start is a negative number, the starting point begins at the end of the array, where -1 is the last element. “end” is a number specifying the index of the ending point for the slice. If you omit this parameter, the slice includes all elements from the start to the end of the array. If end is a negative number, the ending point is specified from the end of the array, where -1 is the last element. Here is an example:

```
name = ['zak', 'ellen', 'alex', 'malachay']
name1 = name.slice(0, 2);
trace(name1);
name2 = name.slice(0);
trace(name2);
name3 = name.slice(-1);
trace(name3);
name4 = name.slice(0, -1);
trace(name4);
```

Figure 5.5 shows the Output window for the above results.

Adding and Inserting Elements

The push() method gets its name from the programming concept of a stack. A stack can be seen like a stack of cards. When you add a card to the top you are in effect pushing the card in place; all the other cards move down one. When you move a card from the top of the stack you are doing what is commonly known in the programming world as last-in-first-out (LIFO); this method is typically used for history lists, like the back button on your browser. The first-in-first-out (FIFO) stack is more equal and works on a first come, first served basis. It works like people in a train ticket line: as you move to the front you get attended to and when satisfied you then leave the line to get on with your journey. So what happens with FIFO is that the first element of the array is

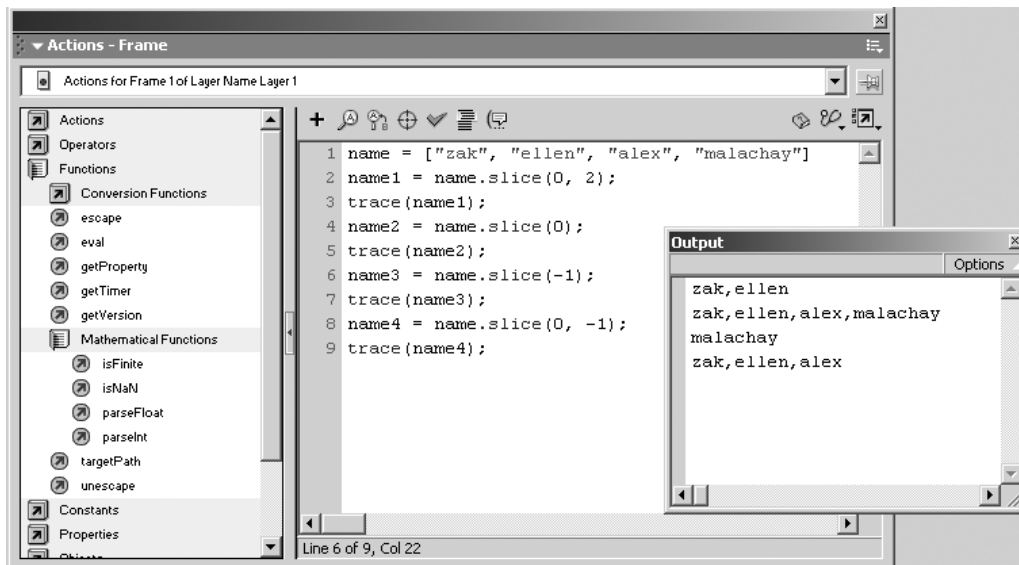


Figure 5.5 Output window for the slice example

Table 5.1 Method summary for the Array object

Method	Description
<code>Array.concat</code>	Concatenates the parameters and returns them as a new array
<code>Array.join</code>	Joins all elements of an array into a string
<code>Array.pop</code>	Removes the last element of an array and returns its value
<code>Array.push</code>	Adds one or more elements to the end of an array and returns the array's new length
<code>Array.reverse</code>	Reverses the direction of an array
<code>Array.shift</code>	Removes the first element from an array and returns its value
<code>Array.slice</code>	Extracts a section of an array and returns it as a new array
<code>Array.sort</code>	Sorts an array in place
<code>Array.sortOn</code>	Sorts an array based on a field in the array
<code>Array.splice</code>	Adds and/or removes elements from an array
<code>Array.toString</code>	Returns a string representing the elements in the Array object
<code>Array.unshift</code>	Adds one or more elements to the beginning of an array and returns the array's new length

actioned and then removed, and the next element moves up. The word `push` implies that you are using the LIFO stack and the word `append` implies that you are using the FIFO stack. In either case, elements are added to the “end” of the stack.

The array object provides the `pop/push` and `shift/unshift` methods to add and remove elements from the array. The `pop` method “pops” an element from the end of the array and returns it, while the `shift` method “shifts” an element from the start of the array and returns it. Their opposites are

used to add elements to the array. The push method is used to “push” one or more elements onto the end of the array, while unshift is used to add one or more to the start. These can save time and provide a nice alternative to indexes. For example, the following code moves the elements from one array to another and reverses the order of the values:

```
arr1 = new Array(0,1,2,3,4,5,6);
arr2 = new Array( );
trace(''arr1 = '' + arr1);
while (arr1.length>0) {
    // pop() removes and returns the last element
    arr2.push(arr1.pop( ));
}
trace(''arr2 = '' + arr2);
```

Arrays can also be useful for defining instances of movie clips – for example, download from www.sprite.net/understanding the following (generate.fla):

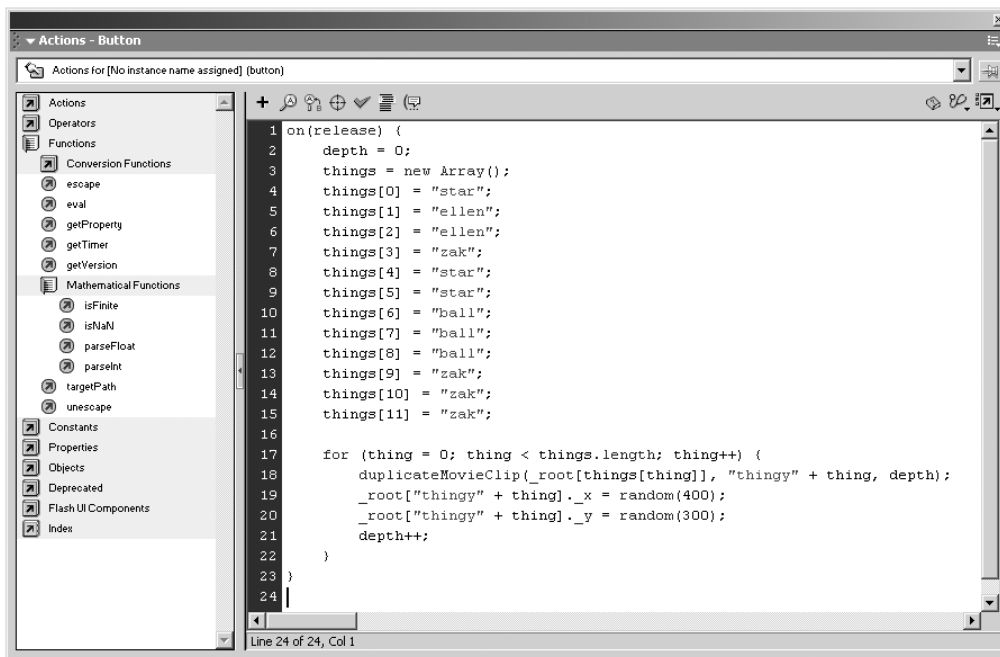


Figure 5.6 Adding the code to the Generate! button object action

This code defines a new array called “things”; each element of the array contains the name of a movie on the stage. The “for” loop cycles through each element of the “things” array and creates a copy of the particular movie referenced at that element in the array, i.e. on the fourth loop, the “zak” movie will be duplicated since it is the fourth element in the “things” array. Once duplicated, the new copy is randomly placed on the stage.

Array Manipulation Methods

ActionScript has many built-in methods for reordering, sorting elements, converting array elements to strings and extracting arrays with other arrays. The `reverse()` method does exactly as its name suggests – it reverses the order of the elements within the array.

```
myArray.reverse()
so the example of
a = new Array('Zak','Alex','Ellen')
a.reverse();
trace(a); displays 'Ellen,Alex,Zak'
```

As an example we use `reverse()` to reorder a sorted list.

The `sort()` and `sortOn()` methods rearrange elements in an arbitrary manner that we provide. Sorting an array alphabetically is really easy – here is an example:

```
MyArray.sort()
```

This method with no arguments is temporarily converted to strings and sorted according to their Unicode code points:

```
var name = ['Zak','Alex','Ellen'];
name.sort();
trace(name); //the display is Alex,Ellen,Zak
```

You can create rules for the `sort()` method of your own choosing, but be aware when sorting out alphabetical lists that upper case comes first. So expect to see upper case Z precede lower case a.

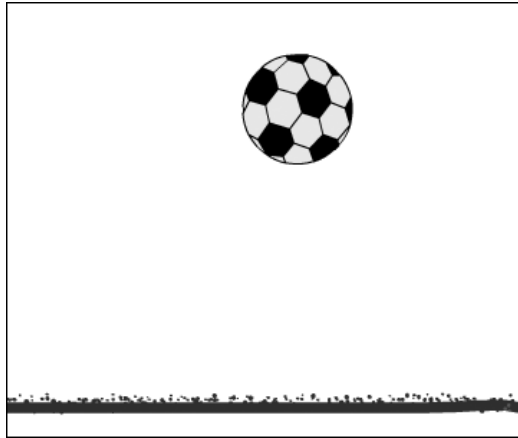
ToString

The `trace()` method automatically converts all arguments to string values. When `trace()` converts an array to a string, it applies the `toString()` method. It converts the elements of the array to string values and then joins them together as a comma-delimited list, as in this example:

```
age = new Array(20, 25, 30, 35);
trace(age.toString()); //displays 20,25,30,35
```

Arrays are amongst the simplest and yet most useful objects in ActionScript. What we have covered in this chapter is simply an introduction to using arrays; what you need to do now is practice creating and working with arrays. It will not be long before you look back and wonder how you coped without them.

This Page Intentionally Left Blank



OBJECT FUNDAMENTALS

03

This Page Intentionally Left Blank

6 Keyboard Object

In this chapter we will look at how to listen to keyboard input with and without creating an object and a constructor. We will use the methods of the Key object to build an interface that can be controlled by a user with a standard keyboard. The properties of the Key object are constants representing the keys most commonly used to control games. For a complete list of key code values, see Appendix B at the end of this book. We will first look at the conventions within ActionScript and touch on the issues of Object-Oriented Programming. We will then move to a few tutorials on keyboard input.

The Structure of ActionScript Code

Only keywords in ActionScript are case sensitive; with the rest of ActionScript, you can use upper and lower case letters interchangeably. ActionScript is not fully case sensitive and therefore it deviates from the ECM-262 standard. You can in theory write code in any case, although only one case will color code in the Actions panel. Unlike variables and construct names, stored values are case sensitive. So if the variable or instance name is mySprite, it is the same as MYSPRITE or mysprite. But the following script is not true:

```
''MYSPRITE'' == ''mysprite''
```

This is because the “==” is a string value comparing values to one another and it is case sensitive with string values.

The function is a keyword and should not be capitalized. If you have correct case in your ActionScript the piece of code turns blue.

It is good practice to follow consistent capitalization conventions, such as those used in this manual, to make it easier to identify names of functions and variables when reading ActionScript code. Because ActionScript is not case sensitive, you must not use variable names that match built-in ActionScript objects. For example, the following is not allowed:

```
date = new Date( );
```

Instead, use the variable names myDate, theDate, and so on. If you don't use correct capitalization with keywords, your script will have errors.

Using ActionScript Syntax – A Revision

ActionScript has rules of grammar and punctuation that determine which characters and words are used to create meaning and in which order they can be written. In English, a period ends a sentence. In ActionScript, a semicolon ends a statement.

The following general rules apply to all ActionScript. Most ActionScript terms also have their own individual requirements.

Curly braces

Over the last few chapters you will have used the curly braces to separate blocks of code. ActionScript statements are grouped together into blocks with curly braces ({ }), as in the following script:

```
on(release) {  
    myDate = new Date( );  
    currentMonth = myDate.getMonth( );  
}
```

Semicolons

An ActionScript statement is terminated with a semicolon, but it is only a convention that suggests you end your statements with a semicolon, they are not strictly required.

```
date = passedDate.getDay( );  
var y = 5;  
  
date = passedDate.getDay( )  
var y = 5
```

The above two pieces of code specifying two variables will both compile without errors. So, if you omit the terminating semicolon, Flash will still compile your script successfully. However, using semicolons is good scripting practice, as they indicate to other readers where your code is terminated for that chunk of code. It is a bit like writing a sentence without punctuation; your reader might misunderstand what you are trying to communicate. In some instances it could affect your results if you have not got used to terminating your code. Take, for example, this statement:

```
function removeOne(sprite){  
    return  
    sprite -- 1  
}
```

So instead of return giving the value `-1`, the function will return undefined because the return statement is legal. In the case of return statements the keyword should never be separate from its expression. In fact, it should look like this:

```
function removeOne(sprite){
    return sprite - 1;
}
```

Note that a semicolon terminates individual statements but is not required where a statement ends in a curly brace.

```
on(release) {
    myDate = new Date( );           // no semicolon here
    currentMonth = myDate.getMonth( ); // semicolon here
}                                   // no semicolon here
```

Parentheses

When you define a function, you must place any parameters inside parentheses:

```
function myFunction (author, book, chapter){
    ...
}
```

When you call a function, include any parameters passed to the function in parentheses, as shown here:

```
myFunction ('Steve', 10, true);
```

Parentheses `()` can also be used to group code to override precedence of an operator to make your ActionScript statements easier to read. You also use parentheses to evaluate an expression on the left side of a dot in dot syntax. For example, in the following statement, the parentheses cause `new Color(this)` to evaluate and create a new `Color` object:

```
onClipEvent(enterFrame) {
    (new Color(this)).setRGB(0xffffffff);
}
```

If parentheses hadn't been used, you would need to add a statement to evaluate the expression:

```
onClipEvent(enterFrame) {
    myColor = new Color(this);
    myColor.setRGB(0xffffffff);
}
```


Comments

In the Actions panel, use comments to add notes to scripts, these comments are ignored by the interpreter and are specifically for the benefit of individuals needing to read the code. Comments are useful for keeping track of what you intended, and for passing information to other developers if you work in a collaborative environment or are providing samples. Comments come in two flavors. The first is `//`, which comments out the line. The second is `/*`; this comments out all the text until you close the tag, like this `*/`. The later version is described as multiline commenting and is only available in Expert Mode in the Actions panel or in an external “.as” source file. Comments allow us to temporarily disable code for testing purposes.

When you choose the comment action, the characters `//` are inserted into the script. Even a simple script is easier to understand if you make notes as you create it:

```
onClipEvent(load){

    // defining the distance the mc will move to when the arrow
    // keys are pressed
    moveDist=50;

    // defining the target coordinates of this mc
    tx=_x;
    ty=_y;
}
```

The multiline commenting looks like this:

```
/*.....stop this code from running

onClipEvent(load){
    // defining the distance the mc will move to when the arrow
    keys are pressed
    moveDist=50;

    // defining the target coordinates of this mc
    tx=_x;
    ty=_y;
}
*/
```

Colored Syntax is turned on in the Actions panel, comments are pink by default in the Script pane. Comments can be any length without affecting the size of the exported file, and they do not need to follow rules for ActionScript syntax or keywords.

Keywords

ActionScript reserves words for specific use within the language, so you can't use them as variable, function or label names. These are reserved for use by the interpreter and will cause errors in the code if we try to use them. Table 6.1 lists all ActionScript keywords.

Table 6.1 *ActionScript keywords*

break	else	instanceof	typeof
case	for	new	var
continue	function	return	void
default	if	switch	while
delete	in	this	with

In addition to the defined reserved keywords, you should also avoid using the names of the built-in properties, methods and objects.

Constants

A constant is a property whose value never changes. For example, the constants BACKSPACE, ENTER, QUOTE, RETURN, SPACE and TAB are properties of the Key object and refer to keyboard keys. To test whether the user is pressing the RIGHT arrow key, you could use the following statement:

```
// the arrow keys change the coordinates of this mc

if (Key.isDown(Key.RIGHT)) {
    tx+=moveDist;
}
```

An Outline of Object-Oriented Programming

Nearly everything in Flash is represented by an object, so if it is a button, text field or movie clip it is an object. Even if you have no grasp of Object-Oriented Programming (OOP) you will have to use object technique to use the buttons, control movies and as in the tutorial at the beginning of this chapter to read keyboard input.

In object-oriented scripting, information is organized into classes. A class is an organizational label. So a vehicle is a class which contains a number of vehicles; these can be a motorbike, van or a car object. The additional and very important feature of a class is that a hierarchy exists within a class and this hierarchy inherits qualities from its higher level object. You can create multiple instances of a class, called objects, to use in your scripts. You can create your own classes and use the built-in ActionScript classes; the built-in classes are located in the Objects folder of the Actions panel.

When you create a class, you define all the properties (characteristics) and methods (behaviors) of each object it creates, just as real-world objects are defined. For example, a vehicle could be said to have properties such as type, passenger number and color, and methods such as speed and sound. In this example, vehicle would be a class, and each individual vehicle would be an object, or an instance of that class.

You can see objects as containers of data, or they can be graphically represented on the Stage as movie clips, buttons or text fields. All movie clips are instances of the built-in class `MovieClip`, and all buttons are instances of the built-in class `Button`. Each movie clip instance contains all the properties.

ActionScript provides many built-in classes; a movie clip is a class or a text field is a class. What we have to do is create individual instances of each of these classes. To create an instance of an object you need to instantiate the object. To do this we use the new operator with the constructor function. This initializes the object.

```
myListener = new Object( );
```

The following code creates a `Date` object and stores the reference to it in the variable:

```
var myDate = new Date( );
```

Throughout this chapter we have used the `Key` object without much consideration to OOP. The following chapters will look at the color and sound object and again beyond the basics of creating an object we are not going to delve into the world of OOP. You can learn a great deal about programming within a Flash environment by sticking to the basic concepts. In time, as you become more confident in your programming ability, then OOP is definitely an advanced feature that will help you develop your skill. You can build some very sophisticated content without needing to grapple with the issue of inheritance and prototyping in an object environment and so beyond this very basic introduction to OOP we will not cover OOP principles but will continue to talk about objects and constructors.

The Key Object

The `Key` object is used to determine which keys are currently depressed and the last key to be released. Not all keyboards are identical, but there are two approaches for detecting keyboard activity:

- We can use the `isDown()` method, which is particularly good for states that need constant polling.
- We can check which key was last released using the `getCode()` and `getAscii()` methods. This is by far the most common method to use. With this approach you do not need to constantly poll for any activity.

The Windows virtual keycode returned by the `getCode()` method and required by the `isDown()` method is a number representing the physical keys on the keyboard and not their ASCII values. The keycodes of the keys A to Z are the same as the code points 65 to 90 for the equivalent upper case Latin 1 letters, which also happens to be the same as their ASCII codes. The keycodes for 0 to 9 have the values between 48 and 57 and likewise are the same for Latin 1 and ASCII. Most of the other keys are non-number and non-letter and are available as properties. The following code is a fragment from our example later on in this chapter:

```
if (Key.isDown(key.UP)) {
    trace('The arrow key is being pressed')
```

The Key object has six methods to enable enhanced keyboard functionality, outlined below.

getAscii()

This returns the decimal ASCII value of the current Key object. So if you press P, then `key.getAscii()` will return 80. This method does not accept any arguments.

getCode()

Returns the keycode value of the last key pressed. The keycode is usually the same for both upper and lower case values.

isDown()

Returns true if the key specified in keycode is pressed. The `isDown()` method returns a Boolean value (true or false).

isToggled(keycode)

Returns true if the Caps Lock or Num Lock key is activated (toggled). You can specify the value as either keycode, `key.isToggled(key.CAPSLOCK)`, or as key constants, `key.isToggled(20)` return true if the Caps Lock key is enabled (toggled). The keycode values for the Caps Lock and Num Lock keys are identical on the Macintosh platform.

addListener(listener)

Registers an object to receive `onKeyDown` and `onKeyUp` notification. When a key is pressed or released, all listening objects registered with `addListener` have either their `onKeyDown` method or `onKeyUp` method invoked. Multiple objects can listen for keyboard notifications. The following script creates a listener object called `myListener` and attaches a function to each of the up and down state of a key press:

```
myListener = new Object( );
myListener.onKeyDown = function ( ) {
    trace('You pressed a key.');
```

```

myListener.onKeyUp = function ( ) {
    trace ( ' 'You released a key.' ' );
}
Key.addListener(myListener);

```

The **Listener.onKeyDown ()** method allows you to detect any downward stroke key pressed.

The **Listener.onKeyUp ()** method allows you to detect when a key has been released.

removeListener(listener)

If the listener was successfully removed, the method returns true. If the listener was not successfully removed – for example, if the listener was not on the Key object's listener list – the method returns false.

Table 6.2 *Property summary for the Key object (all of the properties for the Key object are constant)*

Property	Description
Key.BACKSPACE	The keycode value for the Backspace key (8)
Key.CAPSLOCK	The keycode value for the Caps Lock key (20)
Key.CONTROL	The keycode value for the Control key (17)
Key.DELETEKEY	The keycode value for the Delete key (46)
Key.DOWN	The keycode value for the Down Arrow key (40)
Key.END	The keycode value for the End key (35)
Key.ENTER	The keycode value for the Enter key (13)
Key.ESCAPE	The keycode value for the Escape key (27)
Key.HOME	The keycode value for the Home key (36)
Key.INSERT	The keycode value for the Insert key (45)
Key.LEFT	The keycode value for the Left Arrow key (37)
Key.PGDN	The keycode value for the Page Down key (34)
Key.PGUP	The keycode value for the Page Up key (33)
Key.RIGHT	The keycode value for the Right Arrow key (39)
Key.SHIFT	The keycode value for the Shift key (16)
Key.SPACE	The keycode value for the Spacebar (32)
Key.TAB	The keycode value for the Tab key (9)
Key.UP	The keycode value for the Up Arrow key (38)

Capturing Keyboard Input

Whenever you press a key, three events occur: keyPress, keyDown, keyUp. The main difference between the keyPress and keyDown events is the way the event handler uses them. The keyPress event is specific to a Button instance. So a script might look like this:

```

On(keyPress ' ' <RIGHT> ' ' ) // actions an event when the right arrow
key is depressed

```

The MovieClip object uses the keyDown event with the onClipEvent() handler, so the script might look like this:

```
onClipEvent(keyDown)
```

A movieEvent is a trigger called an event. When the event takes place, the statements following it within curly brackets are initiated.

The Key Listener object can detect the keyDown event within the onKeyDown() method.

The keyUp event occurs when a key is released and can only be used with the MovieClip object, which detects a keyUp event with the onClipEvent(keyUp) handler and the Key Listener object. The Key Listener object uses keyUp events and the onKeyUp() methods.

Movie Clip Instances

Before we do a walkthrough building a simple keyboard detection application, I just want to go over some of the methods that go to build such an application. The MovieClip detects any key press that occurs and executes the enclosed ActionScript code with that event. The following code executes a simple trace statement when the user presses any key:

```
onClipEvent(keyDown){
    trace('You pressed a key');
}
```

with a slight modification the above code can be used to detect when a key has been released:

```
onClipEvent(keyUp){
    trace('You released a key');
}
```

In our tutorial we will use both methods, but first I would like you to build this very simple movie.

Try the following:

1. Create a new Flash document.
2. Create an empty MovieClip and call it detectMC.
3. Open the Library and drag an instance of the detectMC onto the stage.
4. Name the instance keyDetect.
5. Open the Actions panel, making sure the detectMC is selected on the stage.
6. Insert the following code into the Actions window:

```
onClipEvent(keyDown){
    trace('You pressed a key');
}
```

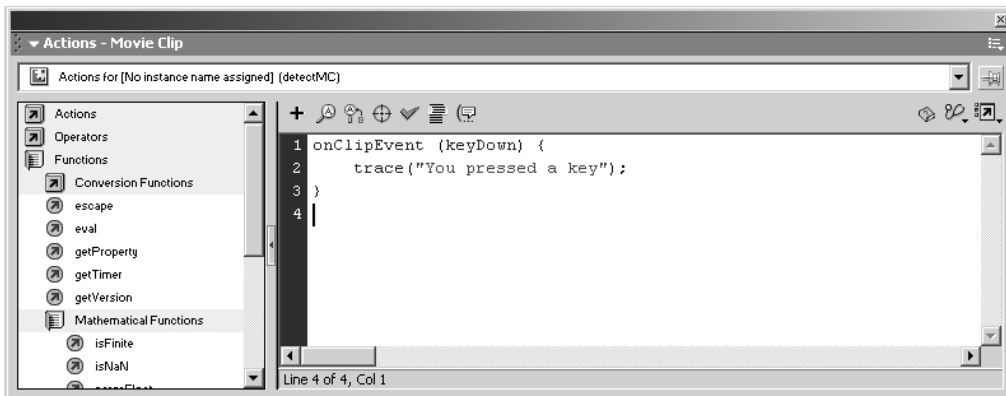


Figure 6.1 *Correctly formatted onClipEvent code*

It should look exactly like Figure 6.1.

7. Save your Flash Movie and call it keyEvent.fla.
8. Now test your movie. The Output window displays the trace message every time you press a key.

Go back to the keyEvent.fla and in the Actions window below the previous onClipEvent type this code:

```
onClipEvent (keyUp) {
    trace('You released a key');
}
```

Now test your movie. The Output window displays the trace message “You pressed a key” every time you press a key and when you release a key it displays “You released a key”. You can download the keyEvent.fla from www.sprite.net/understanding.

If you play a little with the test movie you will see that the events continuously execute if you hold the key down.

Moving Objects with the Arrow Keys

We are going to take a simple object and translate the arrow keys into movements for the object. There are only four directions the arrow keys can point to: UP, DOWN, LEFT, RIGHT. We will follow that through by adding a movement so many pixels in each of the four directions. By making the object a MovieClip and naming it square, we can dictate to it where it has to go on the stage. Here is a fragment of that code:

```
Square._x = The square's x coordinate on the stage
Square._y = The square's y coordinate on the stage
```

Later in this chapter we will demonstrate how we do this using a listener with the `onKeyDown()` method. This listener needs to capture every key press of the arrow keys, creating the basic listener object. But for now we are keeping it simple.

Create a new file with a Square MovieClip on stage. Apply this code to the SquareMC:

```
onClipEvent(load){
    // defining the distance the mc will move to when the arrow
    keys are pressed
    moveDist=50;

    // defining the target coordinates of this mc
    tx=_x;
    ty=_y;
}
```

The first thing to do is to create a number of variables that influence speed and target coordinates for the MC. So we have set ourselves three variables: `moveDist`, `tx`, `ty`. All these variables are initialized once the movie clip is loaded.

```
onClipEvent(keyDown){
    // if it is the "n" key ...
    if(Key.getCode()=='78'){
        trace('DO NOT PRESS THE \'n\' key!')
    }else{
        // return the code of the last pressed key
        trace('the code of the last pressed key:
        '+Key.getCode());
    }
}
```

So what I have done here is to create a key, the “n” key, that returns the message “DO NOT PRESS THE “n” key!”; otherwise it returns the code value of the selected key. The `\'n\’` makes the quotation marks appear in the trace like this “n”.

```
// the arrow keys change the coordinates of this mc
if(Key.isDown(Key.RIGHT)){
    tx+=moveDist;
}
if(Key.isDown(Key.LEFT)){
    tx-=moveDist;
}
if(Key.isDown(Key.UP)){
    ty-=moveDist;
}
```



```
if(Key.isDown(Key.DOWN)){
    ty+=moveDist;
}
}
```

The `isDown` method changes the coordinates by mapping the arrow keys to an action in the relevant direction. The final touch to this is to add some more code just to make the object move smoother:

```
onClipEvent(enterFrame){
    // this code moves the mc smoothly to its target coordinates
    _x+=(tx-_x)/10;
    _y+=(ty-_y)/10;
}
```

The finished file is available for download from www.sprite.net/understanding. The fla name is `keyboardInput.fl`.

The Key Listener Object

The Key Listener object detects any key presses using the `Key` object. Just like the `keyDown` and `keyUp` events for the `onClipEvents()`, the handler detects any key press and so does the `Key` Listener object. The beauty of this object is that no `MovieClip` is necessary to implement the handler. `onClipEvent()` handlers need to be assigned to a `MovieClip` object that is present on stage in order to execute it. Once you create a listener object, you apply “if” and “else” statements to detect the key presses and then implement an action.

```
PressKey = new Object ( );
// you can use the onKeyUp( ) method here

PressKey.onKeyDown = function( ){
    Trace( 'you pressed a key' );
};
key.addListener(PressKey);
```

Create a simple application and apply the above code in addition to the `onKeyUp` handler. Try it:

1. Create a new Flash document.
2. Select frame 1 and open the Actions panel.
3. In Expert Mode type the following:

```
PressKey = new Object ( );
PressKey.onKeyDown = function( ){
```

```

    Trace('you pressed a key');
};
PressKey.onKeyUp = function() {
    Trace('you released a key');

    key.addListener(PressKey);
};

```

The last line of code uses the PressKey object to pass and make active the listener in the last line of code using the addListener method.

4. Save your document and test you movie.

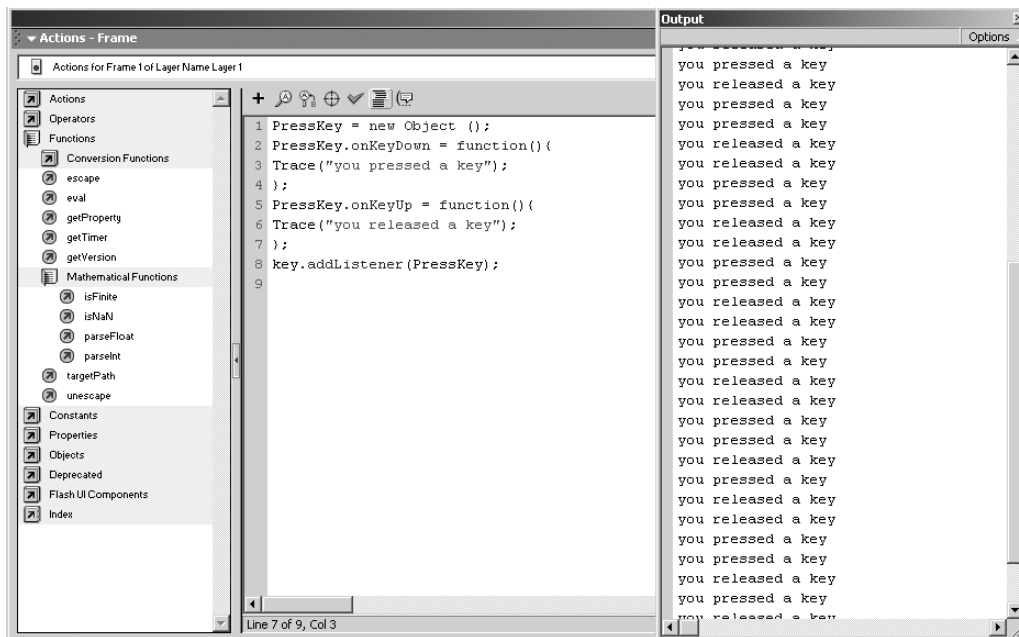


Figure 6.2 Two trace messages appear in the Output window for this example

Determining the Key Pressed using getCode() and Key Properties

In the previous tutorial we did not determine the different keys on the keyboard. All we did was specify whether a key was pressed or released. For this event to be worthwhile in most instances you need to return a value for the key pressed and attach a handler for each different key pressed. Using the getCode() method you can return a value against that event. Similar to the first tutorial in this chapter, we are going to use the Key object to detect right arrow actions.

The getCode() method returns the keycode of the last key pressed. The way it works means that it has a cross platform method for differentiating between key inputs in the different

operating systems. This method can also be used to differentiate between a “6” key pressed on the main keyboard and the “6” key on the numeric keypad. The `getAscii` cannot differentiate between these two and will return the ASCII value of 6. The only downside with this method is it does not differentiate upper and lower case, and it does not detect the modifier keys such as Ctrl and Shift keys.

A short tutorial building a keycode tester is outlined below.

1. Create a new file and type this into it:

```
var tester = new Object( );
tester.onKeyDown = function( ) {
    trace(key.getCode( ));
};
key.addListener(tester);
```

2. Choose Control>Test Movie.
3. Press a key. The Output window will display the keycode value for the key pressed.

Moving a MovieClip using the Key Object

In our first tutorial we looked at moving a square without creating a Key object. In this tutorial we will look at the same movement using the Key object.

1. Create a new Flash file and key into the first frame:

```
pressKey = new Object( );
pressKey.onKeyDown = function( ) {
    if (Key.getCode( ) == Key.LEFT) {
        square._x -= 10;
    }
    if (Key.getCode( ) == Key.RIGHT) {
        square._x += 10;
    }
    if (Key.getCode( ) == Key.UP) {
        square._y -= 10;
    }
    if (Key.getCode( ) == Key.DOWN) {
        square._y += 10;
    }
};
Key.addListener(pressKey);
```

2. Create a movieClip with an instance name of square.
3. Test the movie.

The square moves by 10 pixels in a specified direction with each key press. We have taken advantage of the assignment operator ($+=$) to move the square by 10 pixels. This is applied in both the negative and positive directions, i.e. $+=$, $-=$. We added the key detection code to a Listener object named `pressKey`.

You will have noticed some problem with the changes in movement and direction, especially if you tried to press two keys at the same time. The square moves across the screen in both tutorials because of a feature of your operating system and the keyboard repeat rate. You can see this on both Windows and the Mac OSX:

- **Windows.** Open control panels and double-click the keyboard icon. The properties keyboard dialog box will allow you to change the rate setting to a new value.
- **Macintosh OSX.** Open the system preferences and click the keyboard icon. In the keyboard dialog box change the repeat rate setting to a new value.

Go back and test any of the two tutorials in this chapter and you will notice a change in the speed at which the square object animates across the screen.

setInterval() and the updateAfterEvent()

The `setInterval()` method calls a function or a method or an object at certain intervals while a movie plays. You can use an interval function to update variables from a database or update a time display. If the interval is less than the movie frame rate the interval function is called as close to the interval as possible. If the interval is greater than the movie frame rate, the interval function is only called each time the playhead enters a frame, in order to minimize the impact each time the screen is refreshed.

You must use the `updateAfterEvent` function to make sure that the screen refreshes often enough. The `updateAfterEvent()` method updates the display independent of the frames per second set for the movie. When you call it within an `onClipEvent` handler or as part of a function or method that you pass to `setInterval`, Flash ignores calls to `updateAfterEvent` that are not within an `onClipEvent` handler or part of a function or method passed to `setInterval`.

Here is our move square movie with updated code for the `setInterval` and `updateAfterEvent` methods, with two arguments to execute a function and to set a time between executions:

```
pressKey = new Object( );
pressKey.onKeyDown = enableMove;
pressKey.onKeyUp = disableMove;
Key.addListener(pressKey);
_global.moveInterval = false;

function movesquare( ) {
  if (Key.isDown(Key.LEFT)) {
```

```

        square._x -= 10;
    }
    if (Key.isDown(Key.RIGHT)) {
        square._x += 10;
    }
    if (Key.isDown(Key.UP)) {
        square._y -= 10;
    }
    if (Key.isDown(Key.DOWN)) {
        square._y += 10;
    }
    updateAfterEvent( );
}
function enableMove( ){
    if(!move){
        checkKey = setInterval(movesquare, 10);
        _global.moveInterval = true;
    }
}
function disableMove( ){
    if(move){
        _global.moveInterval = false;
        clearInterval(checkKey);
    }
}

```

You can download this tutorial (astNudge.fla) from www.sprite.net/understanding (and any previous tutorial in this chapter).

This tutorial demonstrates how you can move an object without interference from the key release as the first tutorial and in addition this gives you diagonal movement as you press the horizontal and vertical motion keys at the same time. The time interval established within the `setInterval()` function can be changed: the higher the number, the slower the speed; 10 frames per second (fps) is equal to 100 milliseconds.

```

function enableMove( ){
    if(!move){
        checkKey = setInterval(movesquare, 10);
        _global.moveInterval = true;
    }
}

```

Changing the `_x_y` properties will also change the speed, but if the jumps in pixels are too great then the movie will seem very jerky.

Detecting Key Press Combinations

You need to be able to action multiple key events, in particular when a Ctrl or # key is depressed. Some key combinations may not function in the Flash Player environment or when the movie is playing from within an HTML page. Check for the existence of sensitive key press combinations in the host environment before you finalize your movie. You cannot execute a combination directly from the Key Listener object. You need to create this functionality with a little effort. This involves the use of `setInterval()` to continuously execute a function that checks for the combination keys. Once the keys are detected the `setInterval()` command will be replaced with a `clearInterval()` function. Here is the code for this:

```
function keyDetect() {
    var letter = 'S';
    var pressKey = new String(letter).charCodeAt(0);
    if (Key.isDown(Key.CONTROL) && (Key.isDown(pressKey))) {
        _root.attachMovie('Window', 'Window', 1, {_x: 300, _y:
            200}); disableKeyDetect();
    }
}
```

The `keyDetect()` function checks the current key(s) that is (are) pressed. If both the Ctrl and the Shift keys are pressed, a linked symbol in the movie's library will appear on stage. This symbol displays a dialog box when the combination keys are detected.

```
function enableKeyDetect() {
    if (testKey == 0 || typeof (testKey) == 'undefined') {
        testKey = setInterval(testKey, 50);
    }
}
```

The `enableKeyDetect()` creates a function with a variable named `testKey` that executes a `setInterval()` function specifying the `keyDetect()` function.

```
function disableKeyDetect() {
    for (var i = checkKey; i >= 0; i--) {
        clearInterval(checkKey);
    }
    delete checkKey;
}
```

The `disableKeyDetect()` removes my `setInterval()` calls using the `clearInterval()` function and deletes the `setInterval()` variable `checkKey`.

```
comboKey = {};
comboKey.onKeyDown = enableKeyDetect;
```

```
comboKey.onKeyUp = disableKeyDetect;  
Key.addListener(comboKey);
```

The `comboKey` object enables the key listener and the methods `enableKeyDetect` and `disableKeyDetect`.

7 Sound Object

This chapter looks at the Sound object. The Color and Sound objects are very similar in that they both require separate objects to add new characteristics to a movie clip, such as scripted audio or color changes. This chapter explores the construction of these objects and shows you how to implement them in your movie focusing.

The Sound object lets you control sound in a movie. You can add sounds to a movie clip from the library while the movie is playing and control those sounds. The Sound object has four elements:

- A sound file imported into the library.
- A sound instance created with the new Sound() constructor.
- A MovieClip object that stores the loaded sound file.
- A link to the identifier for the linkage ID.

If you do not specify a target when you create a new Sound object, you can use the methods to control sound for the whole movie. You must use the constructor new Sound to create an instance of the Sound object before calling the methods of the Sound object. You cannot have more than eight Sound objects playing simultaneously.

Table 7.1 *Method summary for the Sound object*

Method	Description
Sound.attachSound	Attaches the sound specified in the parameter
Sound.getBytesLoaded	Returns the number of bytes loaded for the specified sound
Sound.getBytesTotal	Returns the size of the sound in bytes
Sound.getPan	Returns the value of the previous setPan call
Sound.getTransform	Returns the value of the previous setTransform call
Sound.getVolume	Returns the value of the previous setVolume call
Sound.loadSound	Loads an MP3 file into the Flash Player
Sound.setPan	Sets the left/right balance of the sound
Sound.setTransform	Sets the amount of each channel, left and right, to be played in each speaker
Sound.setVolume	Sets the volume level for a sound
Sound.start	Starts playing a sound from the beginning or, optionally, from an offset point set in the parameter
Sound.stop	Stops the specified sound or all sounds currently playing

Table 7.2 *Property summary for the Sound object*

Method	Description
Sound.duration	Length of a sound in milliseconds
Sound.position	Number of milliseconds the sound has been playing

Table 7.3 *Event handler summary for the Sound object*

Method	Description
Sound.onLoad	Invoked when a sound loads
Sound.onSoundComplete	Invoked when a sound stops playing

As with all objects the Sound object requires a constructor to create a new instance of the object. To create a new Sound object you need the following constructor:

```
SoundObjectName = new Sound (MovieClipObjectName)
```

SoundObjectName is the name for the instance of the Sound object you are creating. MovieClipObjectName is the name of the movie clip target/timeline where you want the sound instance to be created. This new sound instance is not the actual sound but is merely a reference to the sound you will be using. The sound is never dragged from the library onto the stage or placed in a frame. The sound is defined in a frame or movie clip and an identifying name is attached to the sound while it resides in the library, except in the case of loading an external MP3, which we will treat separately.

The following script creates an object called mySound that will create a holder for a sound file from the library named Rooster inside a MovieClip named music within a larger MovieClip named musicBox.

```
MySound = new Sound (music.musicBox);
```

You will notice that the above code still does not specify the sound to be used from the library; all this specifies is a holder located in the musicBox timeline. The MovieClip object references an optional argument for the new sound() constructor. If you omit the target, the new sound instance's methods control the sound properties of the movie's main timeline. The following creates a sound resource on the same timeline as the "new Sound()" function call.

Whether you include or omit the target argument, the instance will be created on the timeline that contains the "new Sound()" function call. However, the timeline which this instance's methods will control is determined by the target. There are two concepts here: (1) the familiar concept that object instances (like variables) live in the timeline in which they are

created; and (2) the new concept that Sound objects control the main timeline unless a target is explicitly provided.

```
mySound_1 = new Sound( );
mySound_2 = new Sound( );
mySound_3 = new Sound( );
```

The above code is declared on frame 1 of the main timeline and so sound resources loaded into the Sound object will be stored in the same timeline. If you try to control the Sound object then all the sound files in that MovieClip will respond, as the structure does not allow each sound resource to be stored in its own container. To solve this problem you need to create a unique MovieClip object to hold each sound resource.

In the following tutorial we will create two buttons on stage that control the Sound object. A complete version of this tutorial is available from the website at www.sprite.net/understanding or the sound file ROOSTER.WAV can be downloaded.

1. Create a new document and import a sound file.
2. Choose “Import to Library” from the “File” menu.
3. Import your ROOSTER.WAV.
4. Now right-click on the ROOSTER.WAV library sound object and click on “Linkage...”. The identifier, which, in this example, is ROOSTER, must be unique.
5. In the dialog box “Linkage Properties” gives your sound a unique link name (“Rooster” in our case). Enable “Export for ActionScript” and “Export in first frame” and click “OK”.
6. Create the graphics for the two buttons, one for start and one for stop.
7. Now select the graphics for the start button and choose “Convert to Symbol” from the “Insert” menu (or press F8).
8. In the dialog box give the buttons library names. Make sure you have selected “Button” as Behavior for both buttons and click “OK”.
9. Make one button a stop button and the other a start button. Now that you have the buttons, you have to create a Sound object that will be controlled by the buttons.
10. Create a new layer in the main timeline (_root) and call it script. Select the first frame of it.

```
mySound = new Sound( );
```

This tells the Flash Player that it should create a new Sound object. You then have to attach a sound file to it so there is something to play.

11. Decide where to define the Sound objects. If the Sound objects are all defined on the same level, it simplifies affecting the Sound object later. For example, if the sound is defined at the _root level and then is to be started by pressing a button within a movie clip, the code would appear as follows:

```
on (press) {
    _root.mySound.start( )
}
```

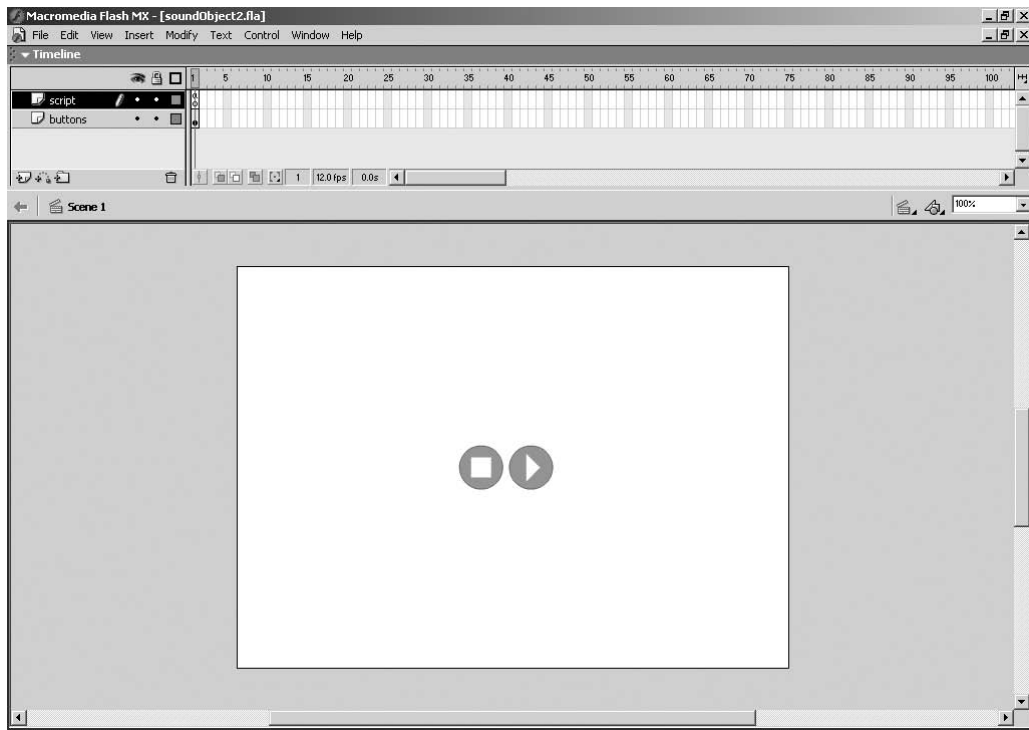


Figure 7.1 *The stage should look like this*

If each Sound object is defined in different movie clips, the path to the Sound object will need to be tracked. When a call is made to the Sound object, the complete path to the location where the Sound object is defined will need to be spelled out in the ActionScript. For example, if the Sound object is defined in a movie clip named “mc1” which is in another movie clip called “mc2”, the ActionScript would appear as follows:

```
on (press) {
    _root.mc1.mc2.mySound.start( );
}

mySound.attachSound( 'Rooster' );
```

The function attachSound uses the unique names of the library objects. Because the Flash Player can’t hear if there is sound playing or not, you need to declare a control variable “soundPlaying” with the value false:

```
soundPlaying = false;
```

12. Now to use the buttons to control the Sound object you need to create two functions – `startSound()` and `stopSound()`:

```
function startSound(){
    if (soundPlaying == false){
        soundPlaying = true;
        mySound.start(0,0);
    }
}
```

```
function stopSound(){
    soundPlaying = false;
    mySound.stop();
}
```

Add this code to your play button:

```
on (release){
    startSound();
}
```

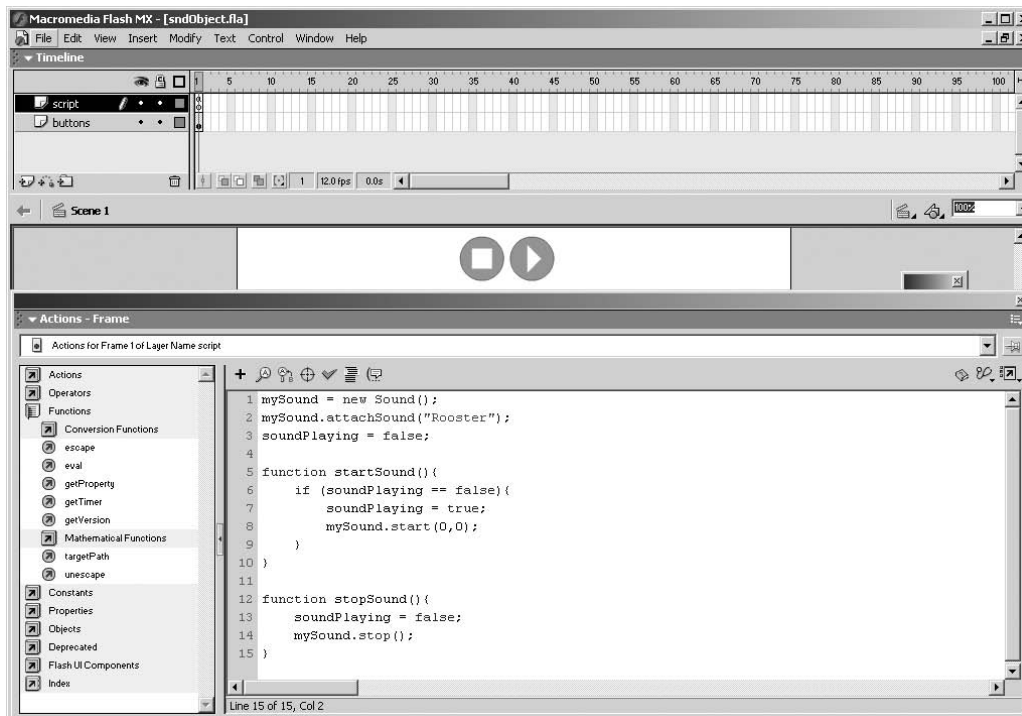


Figure 7.2 The two functions in place in the ActionScript panel

So What is Happening in this Tutorial?

First we will start with the code for the buttons that dictates when a user has clicked on the button:

```
on(release) -
```

The code within the function “startSound()” will be executed. This function first checks if the sound is not playing:

```
if (soundPlaying == false)
```

If so, it sets the variable soundPlaying to true and starts the sound:

```
soundPlaying = true;
mySound.start(0,0);
```

The first value in the brackets is the offset; 0 means the sound is playing from the start. The second value is the number of loops; 0 means no loop.

The code below stops the button:

```
on(release){
    stopSound( );
}
```

The stop button calls the function stopSound() when you clicked on it. The stopSound() function sets the variable soundPlaying to false and stops the sound:

```
soundPlaying = false;
mySound.stop( );
```

You should link only one Sound object to any given MovieClip object. Changes to one Sound object on a MovieClip object will be passed to other Sound objects on the same timeline. In addition, you should never use the name sound for a new Sound object like this:

```
Sound = new Sound( );
```

Organizing Sound Objects

First you have to decide on a naming convention. I allow the instance name of the Sound object to determine all other associated names with that Sound object. This will help later as the ActionScript is written, both in making the names easier to recall and in providing the ability to reuse your code. For example:

- Instance name of sound object: myName.
- Name of its attached audio file's identifier: myNameSnd.
- Instance name of empty movie clip used as a container for Sound object: myNameMc.
- Variable name for position: myNamePosition.
- Variable name for setVolume: myNameVolume.

You should, where possible, define all of your Sound objects on the `_root` level. Define all Sound objects in the first frame of this layer. With this approach you will always be able to see the path to your object. It will be easy to determine the paths to the Sound objects, since they will be like:

```
_root.sound1, _root.sound2, etc.
```

How to Start, Stop and Loop a Sound Object

The syntax for starting a Sound object is as follows:

```
mySound.start([offset,loop])
```

It has two arguments:

```
_root.mySound.start(0,99);
```

In this example, “mySound” is the instance name of the Sound object. The “offset” is the amount of time (in seconds) that is missed before starting to play the sound. The “loop” is the number of times you want the sound to loop. In the above example, this would start mySound at its beginning position of zero and loop it 99 times.

The syntax for stopping a Sound object is:

```
mySound.stop(['ROOSTER'])
```

It has one argument:

```
_root.mySound.stop('ROOSTER')
```

It could also appear as:

```
_root.mySound.stop();
```

In the first example, the linkage identifier “ROOSTER” is used. If the linkage identifier is not used, all sounds will stop – unless the Sound object is defined with a specific movie clip. For example, if you want to stop one of four sounds that are playing, and the sound has not been assigned to a specific movie clip, then you must specify the linkage identifier name as part of the stop command.

If you associate the starting of a sound with an event like a button press, unless you want multiple copies of the Sound object playing at the same time, you need to keep the sound from playing if it is already playing:

```
on (press) {
  if (playing!=true) {
    _root.mySound.start(0,999);
    playing=true;
  }
}
```

And when the sound is stopped by a button:

```
on (press) {
    _root.mySound.stop('ROOSTER');
    playing=false;
}
```

Controlling Independent Sound Objects

Wherever you attach the Sound object, it becomes a child of that movie clip. To affect multiple Sound objects independently, each Sound object must be associated with its own unique movie clip. Any call to a method of a Sound object, such as `setVolume`, will affect all Sound objects that are the child of that timeline. This can be an empty movie clip or one that is being used for something else. The result will be to group your Sound objects by movie clip.

If you have figured out how to fade the volume for a Sound object, but you find that it is also fading the volume for all other Sound objects too. You will find the solution in how you built the sound movie clips.

The following assigns a Sound object as the child of a particular movie clip when the Sound object is initially defined:

1. Define the Sound object:

```
mySound = new Sound(mySoundMc);
mySound.attachSound('ROOSTER');
```

In the above example, “mySoundMc” is the instance name of a movie. You will also notice that there are no quotes around the movie clip instance name.

2. Create an empty movie clip and drag it out onto the `_root` stage. Give the movie clip the instance name of “mySoundMc”. It is not necessary for the “mySoundMc” movie clip to contain the ActionScript that starts the Sound object. The “mySoundMc” movie clip can remain empty. The Sound object “mySound” then becomes a child of “mySoundMc”. The only purpose of mySoundMc is to serve as a container or association for the Sound object.

It is not necessary to create an associated movie clip for Sound objects that only need to be started. It is needed when advanced methods such as `setVolume` and `setPan` are required for multiple independent Sound objects, or when the need arises to stop one Sound object while others continue.

Once this is done, the methods of “mySound” can be manipulated independent of the other Sound objects. The way by which the Sound object’s methods are called do not change. In other words, once you define the Sound object as being attached to a particular movie clip, you never

again need to reference that movie clip to control the Sound object. Since, in our example, the Sound object was defined on the `_root` level, calls to methods of the Sound object would still only need to reference the `_root` path.

So, if you wanted to pan `mySound` completely into the left speaker channel, the following would be used:

```
_root.mySound.setPan(-100);
```

Note the differences in the ways the following Sound objects are defined:

```
mySound = new Sound(mySoundMc);
mySound.attachSound('ROOSTER');

otherSound = new Sound();
otherSound.attachSound('ROOSTER2');

on (press) {_root.mySound.setPan(-100)}
on (press) {_root.otherSound.setPan(100)}
```

If you panned `mySound`, only the sounds associated with the movie clip “`mySoundMc`” would pan to the left. If you panned `otherSound` with the above configuration, all sounds, regardless of where they were defined or to which movie clip they were assigned, would pan to the left.

You can also associate sounds with specific movie clip instances. Define the Sound object as before, except instead of indicating the name of a movie clip instance, use the keyword word “`this`” as follows:

```
mySound = new Sound(this);

mySound.attachSound('ROOSTER');
```

If the Sound object is defined using the keyword “`this`”, calls to the Sound objects, such as `setVolume()` and `stop()`, will only affect the movie clip timeline where the Sound object is defined. The Sound object, `mySound`, which is defined with the keyword “`this`”, reacts in the same way to the Sound object in the previous tutorial when it was associated to a specific movie clip when it was defined on the `_root` timeline.

Using the keyword “`this`” has advantages and disadvantages. One advantage is that, if the SWF file containing the Sound object will be loaded into another movie using `loadMovie`, there is no risk of losing its association with the movie clip it was associated with when it was originally defined. The disadvantage is that the path to the Sound object must be used when calling it into action. If there are a lot of Sound objects scattered about in numerous movie clips, it can be quite confusing.

Controlling all sounds globally is achieved with the following code:

```
mySound = new Sound(mySoundMc);  
mySound.attachSound('ROOSTER');  
  
otherSound = new Sound();  
otherSound.attachSound('ROOSTER');
```

As in the above example, one Sound object was defined on the `_root` timeline as being assigned as a child of the movie clip “mySoundMc”. The other, `otherSound`, was not assigned to any movie clip. Note how these two sounds are stopped in the following examples:

For `mySound`:

```
on(press) {  
    mySound.stop();  
}
```

For `otherSound`:

```
on(press) {  
    otherSound.stop();  
}
```

They are stopped exactly in the same manner. However, because “mySound” was assigned to be the child of the movie clip “mySoundMc”, the stop call to the “mySound” Sound object only stops sounds associated with the movie clip “mySoundMc”. All other sounds continue.

By pressing the stop button for “otherSound”, which was not assigned to a movie clip, it will stop all sounds, including sounds assigned to a specific movie clip. This can be convenient if you want to stop all sounds or control the volume or panning for all sounds. Here is an example from the Flash documentation:

```
globalsound = new Sound();  
  
globalsound.setVolume(50);
```

The above example creates a new instance of the Sound object called `GlobalSound`. The second line calls the `setVolume` method and adjusts the volume on *all* sounds in the movie to 50%.

Volume Control for a Sound Object

In this example, changing the volume for a looping background sound will be described. Unless you will have only one Sound object, or you want to keep all Sound objects at the same volume,

the looping Sound object will need to be associated with a unique movie clip as outlined in the previous section. Here are the steps:

1. Create an empty movie clip, drag it out onto the `_root` stage, and give it an instance name such as “myLoopMc”.
2. Import the sound file ROOSTER from the `sndObject.fla` into the new movie’s library and set its linkage properties identifier to an appropriate name such as “ROOSTER”, assuming it has not maintained all this information from the transfer.
3. Define the Sound object on frame 1 of a layer – for example, called “Sound Objects” using an instance name such as “myLoop” – and use the `setVolume` method to establish the volume level. A variable, “myLoopVolume”, will be used to set the volume, so that it can be manipulated later.

The resulting Sound object should appear as follows:

```
myLoop = new Sound(myLoopMc);
myLoop.attachSound('myLoop01');
myLoopVolume=50;
myLoop.setVolume(myLoopVolume);
```

Volume for a Sound object can be set between 0 and 100. In the example, the volume is set to 50%. The `setVolume` can be placed in a variety of locations – on a button, in a function or on a frame within another movie clip on stage. Just remember that if it’s placed anywhere besides the `_root` level, the path to where it was defined needs to be included as follows:

```
_root.myLoop.setVolume(myLoopVolume)
```

If you wanted to include a volume slider, create a movie clip that can be dragged as in the following tutorial. Create a formula to have either its *x*- or *y*-axis values control the `setVolume` variable. As the axis increases or decreases, the volume percentage increases or decreases an equal amount.

Building a Horizontal Volume Slider

1. Create two images: the slider track and the image that will slide. Make the base of your slider 100 pixels wide, as the width will then correspond exactly to the volume range.
2. Place both on stage as movie clips.
3. From the pull-down top menu, select “View” and then “Rulers”.
4. Place your slider image exactly in the middle of the base. This point will represent volume at 50%.
5. Give your slider movie clip the instance name of “hslider”.
6. Select your slider movie clip, press F9 to access actions for the movie clip and add the following:

```

// the slider moves 50 pixels to the left

onClipEvent(load) {
    left=(_root.hslider._x) - 50;

    // Replace or remove the 1.1 with whatever pixel number
    top=_root.base._y + 1.1;

    //The slider moves 50 pixels to the right.
    right=(_root.hslider._x) + 50;

    // ''bottom'' equals the ''top'' value to remove any vertical
    movement from the slider.
    bottom=_root.base._y + 1.1;

    //Loads the slider in base
    _root.hslider._x=_root.base._width/2;

    //establishes pixel range for calculating volume
    volCalc=_root.hslider._x-50

onClipEvent(enterFrame) {

```

This constantly sets the current *x*-axis position of the slider to “sliderx” and then sets the volume to the *x*-axis value – volCalc.

```

    sliderx=_root.hslider._x;
    myMusicVolume=(sliderx-volCalc);
    _root.myMusic.setVolume(myMusicVolume);
    _root.currentVolume=_root.myMusic.getVolume( );

    //Displays volume in dynamic text box with the Var name of
    currentVolume..

}
onClipEvent(mouseDown) {
    startDrag(this, false, left , top , right, bottom)
}
//
onClipEvent(mouseUp) {
    this.stopDrag( );
}

```

The slider movie clip can now be dragged from its center starting point to both ends of the base image. The goal is to position your slider movie clip graphic so that its x -axis starts directly in the middle of the base. The variable “sliderx” is defined to equal the current x -axis point of your slider movie clip each time a frame is processed – so you can calculate 50 pixels when the slider slides to the right and 50 pixels when it slides to the left.

The ActionScript for the above could also be placed in an event handler on a frame of any movie clip that loops, such as:

```
this.onEnterFrame = function ( ) { code here }
```

Building a vertical slider is the same, but a change has to be made in the way the slider works at it changes from vertical to horizontal sliding. When you drag the slider up, the y -axis pixel number decreases, so the formula used to convert the y -axis to a variable for the `setVolume` method needs to take this into account. The following is the formula used for a vertical slider:

```
// The variable ‘vslidery’ represents the current y-axis
location of the slider (which has the movie clip instance name of
‘vslider’).
```

```
myMusic.setVolume(100-(vslidery-volCalc))
```

Pausing a Sound Object

The Sound object is stopped rather than paused, and at the point it is stopped, its position in milliseconds is recorded as a variable. To continue the Sound object from its “paused” location, the position variable is used in the next start command for the Sound object. The following code works for a pause and continue button:

For the pause button:

```
on (press) {
    myMusicPosition=_root.myMusic.position/1000;
    _root.myMusic.stop(‘myMusicBoom’);
}
```

For the continue or play button:

```
on (press) {
    _root.myMusic.start(myMusicPosition,0);
}
```

For the stop button:

```
on (press) {
    _root.myMusic.stop(‘myMusicBoom’);
    myMusicPosition=0;
}
```

In the above example, “myMusicPosition” is defined as the current position of the Sound object “myMusic” when the pause button is pressed. When the play button is pressed, the starting point for “myMusic” is set to myMusicPosition. When the stop button is pressed, the variable “myMusicPosition” is reset to zero.

Starting Actions when a Sound Object Ends

The method “onSoundComplete” allows you to have actions occur upon completion of a Sound object. The syntax for this for a Sound object with the instance name of “myMusic” is as follows:

```
myMusic.onSoundComplete = callbackFunction
```

Commonly, “onSoundComplete” is used as follows:

```
myMusic.onSoundComplete = function( ) {
    trace( 'My Music has finished' );
}
```

If you wanted the next dynamically loaded MP3 to load when myMusic completes, the following code could be used:

```
myMusic.onSoundComplete = function( ) {
    _root.myNextTrack.loadSound( 'myMusic.mp3' );
    _root.myNextTrack.start( );
}
```

In the above example, a Sound object with the instance name of “myNextTrack”, which was defined on the _root timeline, would load the music file defined at that point and start playing it as soon as the file “myMusic.mp3” completed.

The onSoundComplete method does not need to be called in a movie clip that loops. Once the frame containing the onSoundComplete code has been processed, the function will process its commands every time the sound completes. In fact, if the onSoundComplete method is placed in a one-frame loop it may not work, as the loop is constantly redefining the function.

If the sound starts and ends a second time, the onSoundComplete call will continue activating whatever ActionScript was contained within its function definition. If you don’t want the function to call the initial onSoundComplete actions, redefine the onSoundComplete with a new function.

If a Sound object is started while the same Sound object is playing, the Sound object will play on top of itself. This is not a way to keep the play button from restarting the sound while it is playing. Most of the examples set a variable of “playing=true” when the play button is pressed. The play button also looks to see if “playing!=true” before allowing the play code to process.

However, when the Sound object completes, the “playing” still equals “true”, which keeps the play button inactive even when the sound has completed.

One way to resolve this is to use the `onSoundComplete` method. The following code is used for the play and stop buttons to build the example used in this section:

The play button:

```
on (press) {

    //Play Button
    if (playing!=true) {
        _root.firstSound.start( );
        playing=true;
        _root.myTextBox=' 'Playing' '
    }
    _root.firstSound.onSoundComplete = function( ) {
        _root.myTextBox=' 'Complete' '
        playing=false;
    }
}
```

The stop button:

```
on (press) {
    if (playing==true) {
        _root.firstSound.stop(' 'firstSound01' ');
        playing=false;
        _root.myTextBox=' 'Stopped' ';
    }
}
```

In previous versions of Flash, timing an animation to end at the same time a sound ended was very difficult to achieve. With this method, there is a way to make sure that animation and sound synchronize. Combine `onSoundComplete` with the new “position” method of the Sound object, and your animation to sound synchronization possibilities are so much easier and more controlled.

Loading an External MP3 Dynamically as a Sound Object

The “loadSound” method of the Sound object allows MP3 files to be loaded dynamically. Most of what has already been described about Sound objects is true for dynamically loaded Sound objects. The notable exceptions involve loading the Sound object as streaming versus as an event. If the Sound object is loaded as streaming, it begins to play before it has completely downloaded

onto the user's computer. If the Sound object is loaded as an event, the file must load completely before it can be played.

The syntax for dynamically loading a Sound object with an instance name of “myMusic” is as follows:

```
myMusic.loadSound('url', isStreaming)
```

The Sound object method, “loadSound”, is used as follows:

```
myMusic.loadSound('sample.mp3', true)
```

In the above example, the MP3 file, sample.mp3, is loaded from the same folder that contains the SWF file. It loads into the Sound object “myMusic”, and it is loading as a streaming Sound object. Loading as an event is the default. If neither true nor false is specified for streaming, the Sound object will load as an event.

To use the loadSound method, the Sound object must still be defined, the main difference being that it is not attached to a sound in the library. The following is a common way to define the Sound object with the loadSound method:

```
myMusic = new Sound(myMusicMc);  
myMusic.loadSound('music.mp3');
```

In the above example, a new Sound object is defined with the instance name of “myMusic”, and will load as a child of the movie clip with an instance name of “myMusicMc”. This will give you the ability to control its properties independent of movie clip sounds on other timelines.

In the second line, the external file “sample.mp3” is instructed to load into the Sound object “myMusic”. It will load as an event, versus streaming, as the Boolean value of true or false is not specified. Therefore, the default status (event) will be used.

All of the controls for Sound objects described thus far work for dynamically loaded streaming MP3s, with the exception of starting the MP3 with the mySoundObject.start() method. Streaming MP3s start as soon as there is enough data to play the sound; therefore, the loadSound call to the Sound object is the start command. However, you should be able to stop the streaming sound and restart it with the mySoundObject.start() call. My experience with this is that it is only partially available due to a bug in the Flash Player.

Common Error Message with MP3s

“One or more files were not imported because there were problems reading them.” A common reason to get this error message is that the file being loaded was encoded at over 160 kbps. Flash cannot import MP3s with a bit-rate over 160 kbps.

8 Color Object

The Color object lets you set the RGB color value and color transform of movie clips and retrieve those values once they have been set. The Color object is a special kind of data object used to control the color of MovieClip instances. The Color object works with RGB values, which are also commonly used in web pages to set the color of various elements. RGB stands for Red, Green, Blue and an RGB value is used to describe a particular mix of those three colors onscreen. Flash can understand and display millions of different RGB combinations.

You must use the constructor `new Color()` to create an instance of the Color object before calling its methods.

Creating a New Object

As all with ActionScript objects, the Color objects require a constructor to create a new instance of the object. The constructor creates an instance of the Color object for the movie clip specified by the target parameter. You can then use the methods of that Color object to change the color of the target movie clip. You will need to specify an instance name and a new constructor with a MovieClip instance as its argument:

```
ColorObjectName = new color (MovieClipObjectName);
```

The Color object is called ColorObjectName, and the MovieClipObjectName is the path and name of the MovieClip object you wish to control. So the following:

```
squareColor = new Color(_root.square)
```

tells us that we have a MovieClip named square on the Main timeline (_root), and that a Color object named squareColor is created to control all the colors in the MovieClip. Once a Color object is created, you can use any of the methods of that object's class. Unlike the Mouse and Key objects discussed in the previous chapters, the Color object has specific instances. So you have to invoke their methods through their specific instances, like this:

```
SquareColor.setRGB(0xFF0000)
```

The following example creates an instance of the Color object called myColor for the movie clip myMovieClip and sets its RGB value:

```
myColor = new Color(myMovieClip);  
myColor.setRGB(0xff9933);
```


Table 8.1 *Method summary for the Color object*

Method	Description
Color.getRGB	Returns the numeric RGB value set by the last setRGB call
Color.getTransform	Returns the transform information set by the last setTransform call
Color.setRGB	Sets the hexadecimal representation of the RGB value for a Color object
Color.setTransform	Sets the color transform for a Color object

The four methods in Table 8.1 are the main controls of the Color object. These can be split into two sets. The setRGB() and getRGB() work like the tint setting in the Property Inspector. The setTransform() and getTransform() function affect the advanced setting in the Property Inspector. The first two methods work like the Tint mode of the Property Inspector, while the setTransform() and getTransform() mirror the Advanced mode of the Property Inspector. This mode has eight settings that need to be passed into a transformObject. The following tutorial looks at the Advanced mode of the Property Inspector and how you set your color that will be altered by scripts.

1. Create a new movie.
2. Create a circle and convert it into a MovieClip.
3. With the instance selected, open the Property Inspector.
4. Choose Tint from the Property Inspector drop-down menu.
5. Choose Advanced from the Property Inspector drop-down menu and press the settings button for a settings dialog box.
6. Select the color tints you need. Record the R, G and B values for the colors you wish to use. But do not apply any percentage changes in the Tint mode, as you cannot change this in ActionScript. Record any changed values using the property names shown Figure 8.1, where:

Ra = red percentage
 Rb = red offset
 Ga = green percentage
 Gb = green offset
 Ba = blue percentage
 Bb = blue offset
 Aa = alpha percentage
 Ab = alpha offset

This exercise was looking at how you get your colors. The following tutorials will show you how you apply your colors and amend them using ActionScript.

Making Part of a Color Application

1. Create a new file and then three different shapes on your stage – a square, a circle and a triangle.

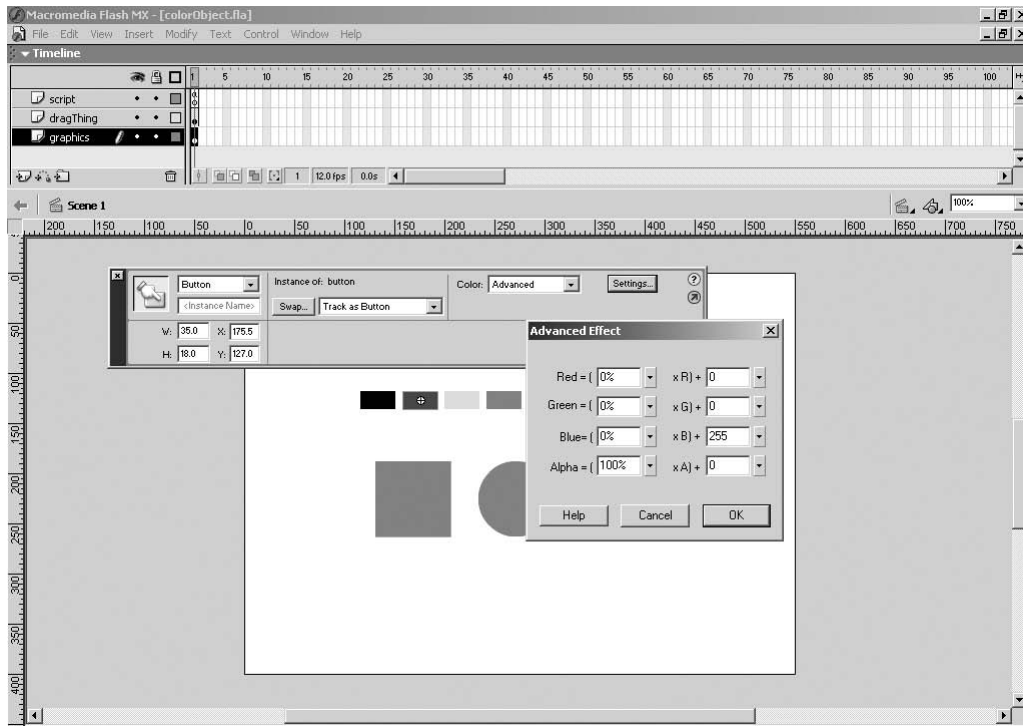


Figure 8.1 *The Advanced Property Inspector drop-down menu and color palette*

- Convert each of them into a movie clip symbol by selecting each shape, choosing Insert > Convert to Symbol, or menu (F8). Give the symbol a library name ("square"), select "movieclip" as behavior and click "OK".
- Next, select the square on your stage and give it the instance name "square" in the "properties" box. Do the same for the circle and the triangle (instance names: "circle" and "triangle").
- Now create a rectangle on your stage and convert it to a button (select it → Insert menu → Convert to Symbol → unique name → behavior: button → OK). Duplicate this button until you have eight in total.
- Create another circle and convert it to a movie clip (as in 2). Give this circle the library name "dragThing". Now select it on the stage and in the properties box give it the instance name "dragThing" and select "Alpha" from the color drop-down. Set the alpha value to 0%.
- The square, the circle and the triangle are the objects we want to give new colors. First we need to choose eight colors that can be used. In this example:

```
(color → rgb values)
black → 0,0,0
blue  → 0,0,255
green → 0,255,0
```

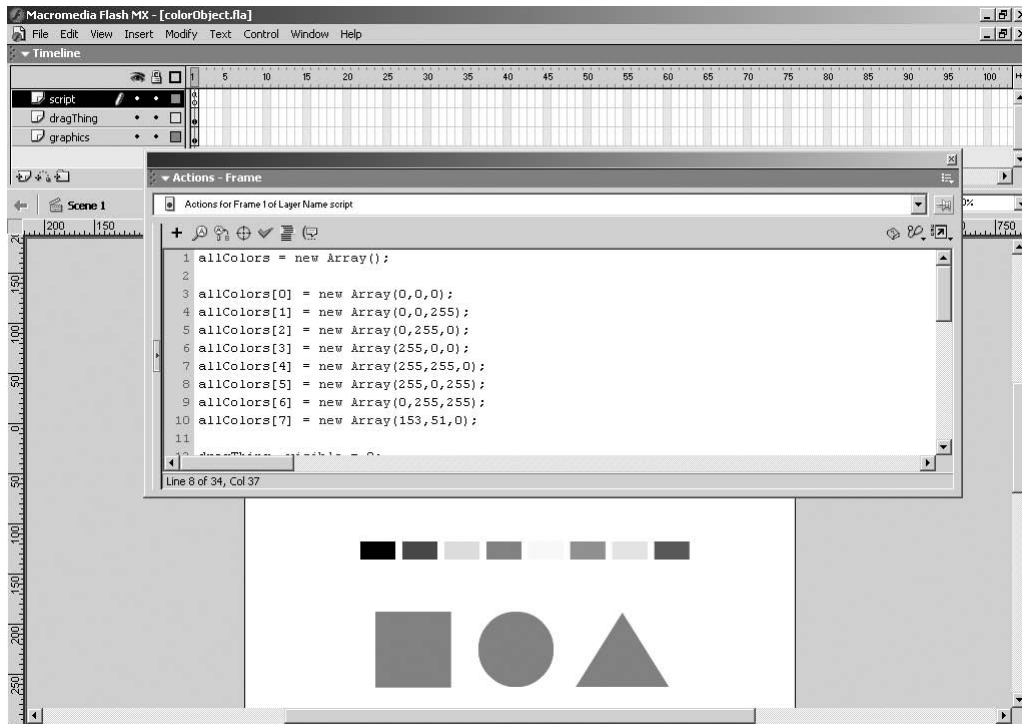


Figure 8.2 *The stage should look like this*

red → 255,0,0
 yellow → 255,255,0
 pink → 255,0,255
 cyan → 0,255,255
 brown → 153,51,0

7. Give each of the buttons one of these colors. Select the button, select from the “color” drop-down in the properties box “Tint” and edit the RGB value referring to the ones above.
8. Now we need to store these color values in the Flash movie. We can do this in an array we call “allColors”. Create a new layer in the main timeline and call it “script”. Select the first frame and enter this code in the actions box:

```
allColors = new Array( );
```

This code creates the array. You can imagine an array as a table:

```

allColors[0] = new Array(0,0,0);
allColors[1] = new Array(0,0,255);
allColors[2] = new Array(0,255,0);
allColors[3] = new Array(255,0,0);
allColors[4] = new Array(255,255,0);

```

```
allColors[5] = new Array(255,0,255);
allColors[6] = new Array(0,255,255);
allColors[7] = new Array(153,51,0);
```

Now this array (table) has one column and eight rows. Each row contains another array with three columns (one for the red value, one for the green, one for the blue) and one row.

9. We add the core functions to the first frame of the main timeline:

```
function dragColor(whatColor){
    whatColor = whatColor - 1;
    dragThing.selectedColor = whatColor;
    dragThing.startDrag(true);
}

function dropColor( ){
    targetMC = dragThing._droptarget;
    targetMC = targetMC.substring(1, dragThing._droptarget.length);
    if (targetMC == 'square' || targetMC == 'circle'
    || targetMC == 'triangle'){
        newValuesRow = dragThing.selectedColor;
        newRedValue = allColors[newValuesRow][0];
        newGreenValue = allColors[newValuesRow][1];
        newBlueValue = allColors[newValuesRow][2];
        stopDrag( );
        myColor = new Color(targetMC);
        myColor.setRGB(newRedValue << 16 | newGreenValue << 8 |
        newBlueValue);
    }
}
```

10. Now add the following code to the first button:

```
on(press){
    dragColor(1);
}
on(release, releaseOutside){
    dropColor( );
}
```

11. When you press on the button the function `dragColor()` will be executed and when you release the mouse button the function `dropColor()` will be executed. This function needs one parameter “whatColor” and that’s the 1 in the brackets.

The dragColor() function

The function `dragColor(whatColor)` is called and (because of the 1 in the brackets) `whatColor` is set to 1. Because array values start counting with 0 instead of 1, we have to decrease the value by 1:

```
whatColor = whatColor - 1; //now whatColor has the value of 0.
```

In the Movie Clip “dragThing” a variable called “selectedColor” is set to the value of `whatColor` (in this case 0):

```
dragThing.selectedColor = whatColor;
```

This allows us to start to drag the `dragThing` MovieClip. The “true” in the brackets means that the center of the MovieClip is on the mouse position.

```
dragThing.startDrag(true);
```

The dropColor() function

Once your mouse button is pressed and you are dragging the MovieClip “dragThing” with the variable “selectedColor” (value: 0), you can move this MovieClip around the stage. When you release the mouse button, the function `dropColor()` is called (this function doesn’t need any parameter).

We dragged the `dragThing` around because we wanted to know if there was a movie clip where we released the mouse button. We first set the variable `targetMC` to the value of the `_droptarget` Property of the `dragThing`. If we released the mouse button over one of our MovieClips (square), the value of `targetMC` will be “/square”:

```
targetMC = dragThing._droptarget;
```

To get the real name of the object we have to cut the first character of the string (“/”):

```
targetMC = targetMC.substring(1,dragThing._droptarget.length);
```

If we released the mouse button somewhere else `targetMC` would be empty. So now we can check if the `droptarget` really was one of our objects (|| means or):

```
if(targetMC == 'square' || targetMC == 'circle' || targetMC == 'triangle'){
```

If so, we read the value of the variable `selectedColor` in `dragThing`. This is the number of the row where the values of the new color are located.

```
newValuesRow = dragThing.selectedColor;
```

Now we read the value for the new color (0 = red value, 1 = green value, 2 = blue value).

```
( allColors[numberOfRow][numberOfCol] )
    newRedValue = allColors[newValuesRow][0];
    newGreenValue = allColors[newValuesRow][1];
    newBlueValue = allColors[newValuesRow][2];
stop dragging the MovieClip dragThing
    stopDrag( );
```

Now for the actual color setting. First we have to create a new Color object. In the brackets we have to set the name of the MovieClip we want to colorize. The name is stored in the variable targetMC:

```
myColor = new Color(targetMC);
```

Bitwise Operators

Now we can give this Color object a new color. Therefore the function setRGB() needs a hexadecimal value that we have to create out of the color values:

```
myColor.setRGB(newRedValue << 16 | newGreenValue << 8 |
    newBlueValue);
```

Bitwise operators are used to explicitly set the bits of numbers – 32 bits in the case of Flash. How these bits are interpreted – float, integer, character – depends on how the variable is used. The exact operation performed depends on the operator, but all bitwise operations evaluate each binary digit (bit) of the 32-bit integer individually to compute a new value.

For example, if the RGB values were 120, 28 and 200 respectively then their binary representations would be:

```
newRedValue    = 120 = 01111000
newGreenValue  = 28  = 00011100
newBlueValue   = 200 = 11001000
```

The setRGB function accepts a single 32 bit integer representing the colour in the format ARGB where each colour is represented by an 8 bit integer with an 8 bit alpha channel, i.e.

```
[ Alpha ] [ Red ] [ Green ] [ Blue ]
00000000 00000000 00000000 00000000
```

To create a colour in this format from the three 8 bit RGB values we need to shift the red and green colours into the correct positions. This is done using the << bitwise shift operator:

Shift Red 16 bits into the correct position:

```
newRedValue << 16
00000000 00000000 00000000 01111000 << 16
= 00000000 01111000 00000000 00000000
```

Shift Green 8 bits into correct position:

```
newGreenValue << 8
00000000 00000000 00000000 00011100 << 8
```

```
= 00000000 00000000 00011100 00000000
```

Blue is already in correct position so no shift required:

```
newBlueValue =
00000000 00000000 00000000 11001000
```

So, by using the OR operator to combine the shifted RGB colour values gives you the correct value for the setRGB function

```
00000000 01111000 00000000 00000000 |
00000000 00000000 00011100 00000000 |
00000000 00000000 00000000 11001000
= 00000000 01111000 00011100 11001000
```

as ActionScript this is

```
((newRedValue << 16) | (newGreenValue << 8) | (newBlueValue))
```

Table 8.2 ActionScript bitwise operators

Operator	Operation performed
&	Bitwise AND
	Bitwise OR
^	Bitwise XOR
~	Bitwise NOT
<<	Shift left
>>	Shift right

To make all the buttons work you have to copy and paste the code from the first button:

```
on(press){
    dragColor(1);
}
on(release, releaseOutside){
    dropColor( );
}
```

and increase the number in the brackets by 1 for each button. For example:

```
button 2: dragColor(2)
button 3: dragColor(3)
```

Now test your movie.

Building an Application with the Color and Sound Objects

In this example you will combine several different Flash features to create a simple coloring application. The tutorial covers the Color, Mouse and Sound objects, Symbol Linkage and the attachMovie action, and setting properties in the Library. But first we are going to have a look at a simple bit of code that is the basis of many applications, in particular gaming applications.

The hitTest Function

You will almost certainly need to use the `hitTest` to detect any kind of collision within an application, and in our application this will be the collision of the cursor/brush with the slider. So here is a quickie on the `hitTest`.

The `MovieClip` object in `ActionScript` provides the `hitTest` function to test for collisions with other movie clips. For example:

```
movie1.hitTest(_root.movie2);
```

The `hitTest` function, in its simplest form, accepts a movie as its single argument. The function will return `true` if `movie1` and `movie2` collide, i.e. the bounding boxes of both movies intersect. The function also provides a way to check if a particular point is located within the movie – for example:

```
movie1.hitTest(100,200,true);
```

In this case, the function will return `true` if the point 100,200 lies within the shape specified by `movie1`. If the third argument is `true` then only the shape defined in the movie is used to detect collisions; if set to `false` then the bounding box of the movie is used.

Download and open `hitTest.fla` from our website on www.sprite.net/understanding, and add the following code to the “zak” movie object action:

```
onClipEvent (enterFrame) {
    if (this.hitTest(_root.ball)) {
        _root.gotoAndPlay('bounce');
    }
}
```

This code tests whether the ball movie has collided with the “zak” movie on every frame. If there is a collision then the movie is moved to the frame named “bounce”.

Now we return to our paint application.

Initializing the Painter Application

A large part of the `ActionScript` for `ZakPainter` is located on the first keyframe of the movie. The `Linkage` property of various symbols will be set to dynamically create instances of the symbols on stage. The movie is made up of three components: initialization, setup, and the creation of four functions that adjust the current color display, check the position of the mouse, play a sound and reset the relevant pictures. You can download a full working copy of `zakPainter.fla` from www.sprite.net/understanding; in addition you can download a copy of the `fla` without the `ActionScript` from the same location.

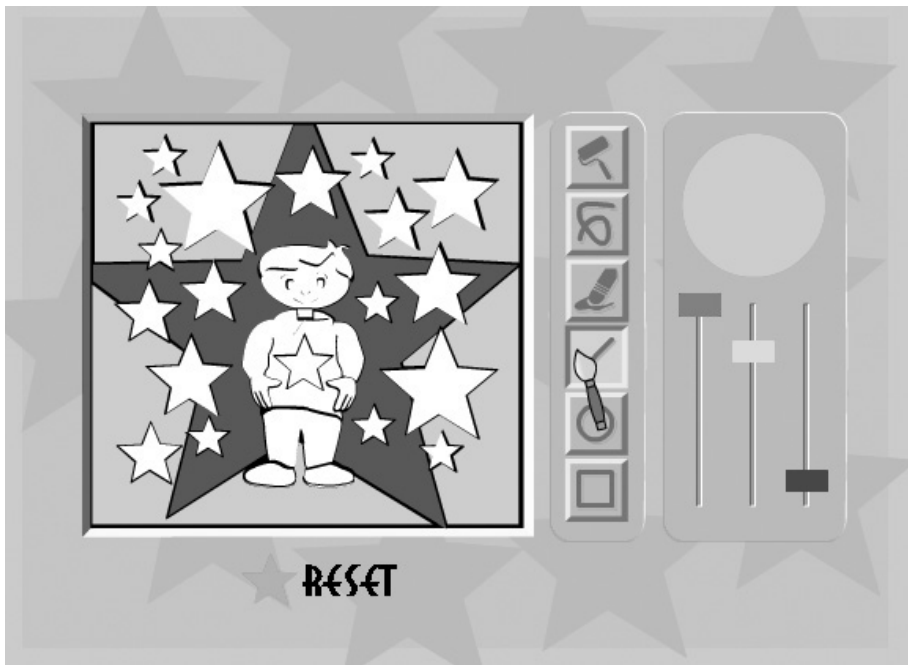


Figure 8.3 *ZakPainter application*

1. Open or download `zakPaintNoScript.fla` and save it to your hard drive. The file contains all the assets in position except for the script.
2. The first task is to attach Movieclip which creates the instance of the arrow key and places it so that it will always appear over the top of other shapes drawn by the user on the stage.
3. In the first keyframe of the ActionScript layer, insert this code:

```
// replace the Mouse Cursor and make the custom cursor follow
your mouse position
Mouse.hide();
_root.attachMovie('CustomCursor', 'CustomCursor',
10000000);
startDrag('CustomCursor', true);
```

4. Add this code in the first frame to create **MyCurrentColor**:

```
// Create Color object for the current color and set the Current
Color Display to this color value
MyCurrentColor = new Color(MyColorDisplay);
SetCurrentColor();
```

5. The movie contains 15 elements that will change color as you paint them, each of which is a different MovieClip instance. To alter their appearances, an instance of the Color object

must be created for each. The first thing we need to do is to create color objects for each picture elements and paint them white.

```
// Create Color objects for each Picture Element
for (PictureElement in this) {
    if (PictureElement.substring(0,7) == 'Element'){
        this[PictureElement+'Color'] = new Color(this
[PictureElement]);
        this[PictureElement+'Color'].setRGB(0xFFFFFF);
    }
}
```

The Color object creates **MyCurrentColor**, which will act as the color controller of the movie clip instance **MyColorDisplay**. **MyColorDisplay** is the larger circle to the top of the color sliders and displays the shade of color that's currently chosen.

We also define in this code block **SetCurrentColor()**, but more on this later. This method works with the three color sliders to change the properties of **MyCurrentColor** to reflect their position.

6. To complete the initialization section of this script, insert the following code:

```
function Reset () {
    // Reset color of each Picture Element
    for (PictureElement in this) {
        if (PictureElement.substring(0,7) == 'Element'){
            this[PictureElement].setRGB(0xFFFFFF);
        } else if (PictureElement.substring(0,5) == 'Shape'){
            this[PictureElement].removeMovieClip();
        }
    }
}
```

7. You also need to create a Sound object on the first frame. **PlaySound(WhatSound)** is the function in the main timeline. Sounds were also added to the tool buttons and the sliders by calling this function.

```
function PlaySound(WhatSound){
    MySound = new Sound(this);
    MySound.attachSound(WhatSound);
    MySound.start(0,0);
}
```

8. The **SetCurrentColor()** function works directly with the `_y` property of the color sliders. The `_y` moving range of each slider is between 182.5 and 310. `_y` can now have values between 0 and 127.5, multiplied by 2: 0-255. You must use the bitwise operations to generate the hex value out of these `_y` values.

```
// Set Color Function
function SetCurrentColor () {
    // Get Color Mix
    RedValue = int(((310-RedSlider._y))*2);
    GreenValue = int(((310-GreenSlider._y))*2);
    BlueValue = int(((310-BlueSlider._y))*2);
    // Set Current Color
    MyCurrentColor.setRGB(RedValue << 16 | GreenValue << 8 |
    BlueValue);
}
```

SetCurrentColor() evaluates the horizontal positioning of each of the three sliders in the color chooser. The sliders visually represent the amount of red, green and blue (respectively) in the current color. So **SetCurrentColor** calculates that mix and changes the appearance of **MyColorDisplay** in response.

The first part of the function creates three variables: **RedValue**, **GreenValue** and **BlueValue**. Each of these variables is assigned a value from 0 to 255, reflecting the **_x** position of its corresponding slider. When all values have been calculated, the resulting color is used to alter the appearance of **MyDisplayColor**. **MyCurrentColor** is the color object attached to that instance, so its **setRGB** method is used to set the shade.

9. The second custom function on this keyframe is **CheckMousePosition()**, which is used to see if the mouse is currently being held over the picture area. The function returns a **true** or **false** value that is used by the drawing tool routines to decide whether or not they should draw. The drawing area starts at the coordinates $x = 40/y = 67$ and ends at $x = 317/y = 323$. This functions checks if the mouse is within these coordinates.

So if the mouse is outside the drawing area, **CheckMousePosition()** will return **false**, and nothing will be drawn. Conversely, if the mouse is over the drawing areas, **CheckMousePosition()** will return **true**, and the operation will continue. Input this code to define the **CheckMousePosition()** function:

```
// Check mouse position for draw operation
function CheckMousePosition () {
    if (_xmouse>40 && _xmouse<317 && _ymouse>67 && _ymouse<323) {
        return true;
    } else {
        return false;
    }
}
```

The Sliding Color Selector

We previously defined **SetCurrentColor()**, allowing the position of the three sliders to be converted into color values. The sliders are just ordinary draggable movie clip instances. In this

section I will use ClipEvents to respond to mouse actions. This is an alternative to using buttons. The instance name of the slider clips are **RedSlider**, **BlueSlider** and **GreenSlider**.

1. Bring up the Actions Panel for the red colored slider handle and add the code for the first ClipEvent handler:

```
onClipEvent (mouseDown) {
    if (this.hitTest(_root._xmouse, _root._ymouse, true)) {
        // play a sound
        _root.PlaySound('Click');
        Dragging = true;
    }
}
onClipEvent (mouseUp) {
    Dragging = false;
}
onClipEvent (enterFrame) {
    if(Dragging){
        if (_root._ymouse < 182.5) {
            _y = 182.5;
        } else if (_root._ymouse > 310) {
            _y = 310;
        } else {
            _y = _root._ymouse;
        }
        _root.setCurrentColor();
    }
}
```

This block of code makes the slider MovieClips active any time the mouse button is clicked over the entire movie. The movie ClipEvents are global and so affect the whole movie.

The **hitTest** can be applied to the slider using the current mouse coordinates. If **hitTest** returns **true**, it is because the user is clicking the slider. In this case, a variable named **Dragging** is set to **true** to indicate that the user is dragging the slider. A sound indicates a selection has been made.

```
onClipEvent (mouseDown) {
    if (this.hitTest(_root._xmouse, _root._ymouse, true)) {
        _root.PlaySound('Click');
        Dragging = true;
    }
}
```

2. Insert this code:

```
onClipEvent (mouseUp) {
    Dragging = false;
}
```

When the mouse button is released the mouse button can no longer be dragging the slider, so **Dragging** is set to **false**.

The **enterFrame** event is a global event and activates every time the Stage is refreshed. So it doesn't matter whether the movie clip it is attached to is playing. (24 fps = refreshes 24 times in a second.)

3. Add the final handler code:

```
onClipEvent (enterFrame) {
    if(Dragging){
        if (_root._ymouse<182.5) {
            _y = 182.5;
        } else if (_root._ymouse>310) {
            _y = 310;
        } else {
            _y = _root._ymouse;
        }
        _root.SetCurrentColor();
    }
}
```

What we are trying to do here is to make the slider behave in a particular way. The desired behavior is that the slider should move with the mouse when clicked and move up or down within a range of 127.5 pixels. So the code in this step alters the position of the slider object to reflect the *y* coordinate of the mouse but only if that position is between *y*:182.5 (the topmost point) and *y*:310 (the bottommost point). If the mouse is outside that range but the mouse button is still pressed, the slider is forced to stay put at either end.

While you are dragging one of the sliders the function **setCurrentColor()** located in *_root* is called every time the stage is refreshed.

This is achieved with a set of **if** statements that allow you to check for two things: first, that **Dragging** equals true, and second, the position of the cursor. The *_y* property of the slider is altered accordingly, being set to 310, 182.5 or the *y* coordinate of the cursor.

The scripts assigned to the red slider are identical to those needed on the other two sliders.

4. Copy the above code that you just entered, select each of the other sliders in turn, and paste the code back into the panel.

5. We now need to define a default drawing tool:

```
MyTool = 'Paint';
```

6. Now for the actual drawing code. It's attached to the MovieClip **MyColorDisplay**. Once this MovieClip is loaded on the stage a variable called **ShapeCount** is set to 0 in this MovieClip:

```
onClipEvent(load){
    ShapeCount = 0;
}
```

This variable keeps a count of all objects that a user draws and is used to give each drawn shape a unique name.

When the user presses the mouse button the function **CheckMousePosition()** we defined in **_root** is called. It checks if the mouse is over the drawing area and returns **true** if it is or **false** if it's not.

When it is in the right position a code for the creation of a shape (depending on the selected drawing tool defined in **MyTool**) will be executed.

If **MyTool** is **Paint** the picture element that is right below the cursor will be colored with our selected color (stored in **MyCurrentColor**).

If **MyTool** is **Eraser** we have to check first what kind of object is below the cursor. If the name of the element starts with "**Shape**" we can remove this object because it's one of the drawn objects. Therefore we can use the **removeMovieClip()**. If its name doesn't start with "**Shape**" it is one of the picture elements and we can color it with white paint to **imitate** an erasing action.

Now for the more complicated Drawing tools. If **MyTool** is **Pen** we just set the variable **PencilDown** to **true** and store the mouse coordinates in the variables **LastX** and **LastY**. I will explain the code that creates the pen line later.

The **Ellipse**, **Rectangle** and **Line** tools work all the same way. So if **MyTool** is one of these, a new MovieClip will be attached to the stage depending on the kind of tool we are using.

The **attachMovie** action is used to dynamically create instances of symbols from the Library, but you must set their linkage properties. So you need to select the specified symbol: choose linkage from the Library's Options menu. In the dialog box, click the Export This Symbol option and type the Identifier name text as highlighted in Table 8.3. The Identifier is always unique for each symbol and is called in the **attachMovie** action.

Table 8.3 *Identifier name text*

Symbol name	Identifier
Movieclip Symbols>Custom Cursor	CustomCursor
Movieclip Symbols>Drawing Shapes>Shape: Ellipse	Ellipse
Movieclip Symbols>Drawing Shapes>Shape: Line	Line
Movieclip Symbols>Drawing Shapes>Shape: Pen Line	Pen
Movieclip Symbols>Drawing Shapes>Shape: Rectangle	Rectangle

The attached object gets an unique name (using the **ShapeCount** variable → “Shape0”, “Shape1”, etc.); it is moved to the coordinates where the cursor is and colored with our currently selected color.

```
onClipEvent (mouseDown) {
    if (_root.CheckMousePosition()) {
        if (_root.MyTool == 'Paint') {
            _root[substring(_root.CustomCursor._droptarget, 2,
50)+'Color'].setRGB(_root.MyCurrentColor.getRGB());
            _root.PlaySound('Fill');
        } else if (_root.MyTool == 'Eraser') {
            if (substring(_root.CustomCursor._droptarget, 2, 5)
== 'Shape') {
                _root[substring(_root.CustomCursor._droptarget, 2,
50)].removeMovieClip();
            } else {
                _root[substring(_root.CustomCursor._droptarget, 2,
50)+'Color'].setRGB(0xFFFFFF);
            }
        } else if (_root.MyTool == 'Pen') {
            LastX = _root._xmouse;
            LastY = _root._ymouse;
            PencilDown = true;
        } else if (_root.MyTool == 'Line' || _root.MyTool == 'Ellipse'
|| _root.MyTool == 'Rectangle') {
            PencilDown = true;
            ShapeCount++;
            _root.attachMovie(_root.MyTool, 'Shape'+ShapeCount,
ShapeCount+100);
            _root['Shape'+ShapeCount]._x = _root._xmouse;
            _root['Shape'+ShapeCount]._y = _root._ymouse;
            _root['Shape'+ShapeCount+'Color'] = new
Color(_root.Shape'+ShapeCount);
            _root['Shape'+ShapeCount+'Color'].setRGB(_root.
MyCurrentColor.getRGB());
        }
    }
}
```

```

    }
  }
}

```

The code for the next ClipEvent handler will be executed every time the stage refreshes. First it checks if the mouse button is pressed and if the mouse is within the drawing boundaries.

If so, we take another look at the selected drawing tool. If it is “Pen” the following happens: we attach a new MovieClip from the library to the stage and give it a unique name (as usual using ShapeCount). Now we give this MovieClip the selected color and place it where our cursor was the last time the stage refreshed. These coordinates are stored in the variables **LastX** and **LastY**. Now we calculate the x -distance and the y -distance from the last cursor position to the current cursor position and scale the movie clip to these values. Now all we have to do is to store the new cursor coordinates into the variables **LastX** and **LastY**. So we are building our freehand pen line by sticking a lot of small lines together.

If we are using the Rectangle, Ellipse or Line tools instead of the Pen tool, we only have to scale the object. We calculate the scale factor also out of the x -position and the y -position of the object and the current cursor position (similar to the Pen tool). For example, if an object ranges from $x = 30$ to $x = 70$ it has a width of 40 pixels.

```

onClipEvent (enterFrame) {
    if (PencilDown && _root.CheckMousePosition()) {
        if (_root.MyTool == "Pen") {
            ShapeCount++;
            _root.attachMovie(_root.MyTool, "Shape" + ShapeCount,
            ShapeCount+100);
            LineColor = new Color(_root["Shape" + ShapeCount]);
            LineColor.setRGB(_root.MyCurrentColor.getRGB());
            _root["Shape" + ShapeCount]._x = LastX;
            _root["Shape" + ShapeCount]._y = LastY;
            _root["Shape" + ShapeCount]._xscale = LastX -
            _root._xmouse;
            _root["Shape" + ShapeCount]._yscale = LastY -
            _root._ymouse;
        } else {
            _root["Shape" + ShapeCount]._xscale = _root["Shape" +
            ShapeCount]._x - _root._xmouse;
            _root["Shape" + ShapeCount]._yscale = _root["Shape" +
            ShapeCount]._y - _root._ymouse;
        }
    }
    LastX = _root._xmouse;
    LastY = _root._ymouse;
}

```


Now when the user releases the mouse button the variable **PencilDown** will be set to false and the code in the enterFrame clip event won't be executed any longer. So the last modified object will stay as it was when the stage refreshed the last time.

```
onClipEvent (mouseUp) {
    PencilDown = false;
}
```

Scripting the Tool Buttons

ZakPainter has six different tools, plus a Reset button. When pressed, each of the tool buttons sets a variable named **MyTool** to a different value to reflect which tool is currently active. The drawing scripts created in the next section control their behavior and then use this text value.

The new drawing API enhances the object-oriented programming power of ActionScript by offering a set of shape-drawing capabilities through the MovieClip object, allowing for programmatic control over the Flash rendering engine.

What are Draw Methods?

Let's put our tutorial on hold and look at the drawing API. The new draw methods are ActionScript commands that are available to every movie clip. These methods allow for the creation of lines and shapes via code at runtime rather than having to rely on pre-drawn graphics. The new methods are:

- MovieClip.beginFill()
- MovieClip.beginGradientFill()
- MovieClip.clear()
- MovieClip.curveTo()
- MovieClip.endFill()
- MovieClip.lineStyle()
- MovieClip.lineTo()
- MovieClip.moveTo()

With these tools you can do just about anything you can do using the Flash MX drawing tools.

The Virtual Pen

Every movie clip has a virtual "pen". This pen is, by default, located at the movie clip's origin (0,0) and has no set stroke or fill, so it is invisible. The draw methods either modify the pen's properties or move the pen. In the movie above, the following ActionScript exists on the button:

```
on (release) {
    _root.lineStyle(1, 0x0000FF, 100);
    _root.lineTo(100, 75);
}
```

Because the `_root` has all the properties of a movie clip, it too has access to the draw methods. The second line establishes that the pen's line should be 1 pixel in width, blue defined as a hexadecimal number, and the line should be 100% opaque.

Depth and Draw Methods

The drawing methods draw behind any content created either manually or via movie clip methods (`duplicateMovieClip` or `attachMovie`) within a clip. For example, when we use the draw methods to draw a line, the line will display behind all of the content that exists in that movie clip. Subsequent lines or fills will appear on top of previous draw method objects, but they will still be behind objects created by other means.

The code for **MyColorDisplay** should look like this:

```
onClipEvent(load){
    ShapeCount = 0;
}

onClipEvent(mouseDown){
    if(_root.CheckMousePosition()){
        if(_root.MyTool == 'Paint'){
            _root[substring(_root.CustomCursor._droptarget, 2,
50)+'Color'].setRGB(_root.MyCurrentColor.getRGB());
            _root.PlaySound('Fill');
        } else if(_root.MyTool == 'Eraser'){
            if(substring(_root.CustomCursor._droptarget, 2, 5)
== 'Shape'){
                _root[substring(_root.CustomCursor._droptarget, 2,
50)].removeMovieClip();
            } else {
                _root[substring(_root.CustomCursor._droptarget, 2,
50)+'Color'].setRGB(0xFFFFFF);
            }
        } else if(_root.MyTool == 'Pen'){
            ShapeCount++;
            _root.createEmptyMovieClip('Shape'+ShapeCount,
ShapeCount+100);
            _root['Shape'+ShapeCount].lineStyle(1, _root.
MyCurrentColor.getRGB(), 100);
            _root['Shape'+ShapeCount].moveTo(_root._xmouse,
_root._ymouse);
            PencilDown = true;
        } else if(_root.MyTool == 'Line' || _root.MyTool ==
'Ellipse' || _root.MyTool == 'Rectangle'){
```

```

        PencilDown = true;
        ShapeCount++;
        _root.attachMovie(_root.MyTool, "Shape"+ShapeCount,
ShapeCount+100);
        _root["Shape"+ShapeCount]._x = _root._xmouse;
        _root["Shape"+ShapeCount]._y = _root._ymouse;
        _root["Shape"+ShapeCount+"Color"] = new
Color("_root.Shape"+ShapeCount);
        _root["Shape"+ShapeCount+"Color"].setRGB(_root.
MyCurrentColor.getRGB());
    }
}

onClipEvent (mouseUp) {
    PencilDown = false;
}

onClipEvent (enterFrame) {
    if (PencilDown && _root.CheckMousePosition()) {
        if (_root.MyTool == "Pen") {
            _root["Shape"+ShapeCount].lineTo (_root._xmouse,
_root._ymouse);
        } else {
            _root["Shape"+ShapeCount]._xscale = _root["Shape"+
ShapeCount]._x - _root._xmouse;
            _root["Shape"+ShapeCount]._yscale = _root["Shape"+
ShapeCount]._y - _root._ymouse;
        }
    }
    LastX = _root._xmouse;
    LastY = _root._ymouse;
}

```

As you can see the code that draws shapes and lines is still the same. We just modified the code that creates the pen line. It works as outlined below.

Our Flash movie has a frame rate of 24 frames per second – this means it refreshes 24 times in a second. When the user presses the mouse button and the mouse is above the drawing area, an empty movie clip is created in our `_root` timeline. We need to create a new movie clip because, as mentioned before, the drawing API draws behind all objects in the targeted timeline. But we are creating a new movie clip in front of all objects and draw in that one. As usual it is given a unique name referring to the `ShapeCount` variable and it is set to the user's mouse position (that's where the

line starts). This is also the registration point for the API. The registration point is the origin of a new line and is updated each time a new line has been drawn. To initialize the style of the line we are drawing we use one of the new drawing API functions **lineStyle(size, color, alpha)**. The size of the line can be between 0 and 255 pixels (0 = hairline). Color has to be a hexadecimal value of the desired color. In our example we are using the color of **MyCurrentColor**. The Alpha value has to be between 0 (invisible) and 100. Within this new empty movie clip our pen line with these given values will be drawn. Once the button is pressed and the movie clip created, the variable **PencilDown** is set to true (as usual too). Now every time the stage refreshes a new line is drawn. If it is the first one the line will start at the position where the user pressed the mouse button (the registration point) to the user's current mouse position. When the line is drawn the registration point is set to the coordinates of the end point of this last line, so the next line will start exactly at this point.

When the user releases the mouse button, the variable **PencilDown** is set to **false** so none of the drawing code will be executed any longer.

How it Works

The ZakPainter application demonstrates the Color object, Flash's built-in mouse tools, and the Flash drawing API. The Color object Instances are created for drawn shapes, picture elements and the current color display object. This approach allows the programmer a simple way of manipulating the movie clip graphics and allows for the creation of many colored copies of the same symbol that can be changed at any time by code driven events or in response to user input.

All colors on the screen are reproduced using a mix of red, green and blue. To describe a color, all you have to do is determine what proportion of red green or blue colors to display.

The Mouse object allows the mouse to hide or show the cursor at will. To create the custom cursor effect, the Mouse object is used to hide the regular arrow/hand cursor. A movie clip instance is then created on-the-fly by using the **attachMovie** action and by scripting that instance to "drag" or follow the mouse movement.

ZakPainter is tied together with a series of ClipEvent handler scripts, applied to movie clip instances. ClipEvents are used here primarily as a substitute for buttons, and they make it possible to create such an application in the most simple way possible. And because different ClipEvent scripts can be added to individual instances of a movie clip, the file can be smaller in memory.

The Color object can be very complicated and this tutorial is as complicated as it gets. Read through it a few times because in there you will find some interesting code that can be reused in lots of different situations.

9 XML Object

What is XML?

XML (extensible markup language) is becoming a standard way to export and import data from different systems. It is a simplified markup language that makes it easy for different systems to read data in, without knowing the exact structure of the data. If the XML data description has been properly created it is possible to add extra information to your XML data files without affecting the systems that are already using the information.

If you load XML data into Flash, an object will be created that will allow you to easily manipulate and extract the data. The same object is available to create XML data, and export it to an external system. XML uses tags very similar to HTML; instead of describing the structure of the page using tags, XML uses the tags to describe the data. This allows a structured data format that can be extended without breaking previous implementations. One of the most significant differences between HTML and XML is that HTML is not extensible. This means that HTML is limited to a set of predefined tags. On the other hand, XML has infinitely many possibilities for tags. In fact, the tags are invented entirely by the author of the document.

In general, information can be loaded into Flash from a variety of sources external to the Flash file. Load variables might work when loading simple or small amounts of information from a text file or CGI. If you have to retrieve differing amounts of information, or it is being created by an external source, you should use XML. XML, however, provides a much more convenient and powerful way to load large amounts of data, as well as data that has a complex and variable structure.

One of the benefits of XML is that it allows for customizable and neat representation of data which can be easily parsed (interpreted by a computer) while at the same time being clearly readable to a human. XML uses “tags” which may or may not include refining elements called “attributes”. HTML is primarily a markup language, while XML – in relation to Flash at least – is primarily a data structure. Consider the HTML bold tag – to create some bold HTML text we use:

```
<b>This is some bold text</b>
```

The simple “tags” at the beginning and end of the text string tell the browser to format that string in a particular manner. XML is similar to the extent that it also uses such tags in its structure; however, in XML, instead of being a way of describing the visual representation of data, tags are used to group related pieces of information into “nodes”. In fact, the above snippet is a properly formatted XML node.

Exchanging and Storing Data

XML has quickly become the standard mechanism for sharing data among different platforms. All that is required is that somebody has written an XML parser for the receiving platform and then that same data can be stored in XML format for use by differing devices.

What Does XML Look Like?

The author controls the structure of an XML document, but it will still conform to some basic rules. With so many parsers for so many devices it is far better to make sure your document sticks to the rules. Following these rules insures that any given XML document will be readable by any parser. In XML, you define tags that identify the type of a piece of data. XML separates the structure of the information from the way it's displayed, so the same XML document can be used and reused in different environments. There are many considerations that can be taken into account for writing well-formed XML; several are very important for working with well-formed XML in ActionScript. The four main considerations are XML declarations, tags, attributes, and DTD.

The Declaration

The first thing that should appear in any well-formed XML document is the declaration. The declaration should look like this:

```
<?xml version='1.1.3'?>
```

This tells you the version number of your document. If you did not include this, ActionScript will cope quite happily without it. But you will get confused as to which version number you are running or another application will read the data and expect to find it there.

XML, unlike HTML, is extremely sensitive when it comes to formatting. An XML document must strictly conform to various rules in order to be processed: this is a good way to enforce clean coding. The rules are as follows.

Tags

Tags in XML can be anything you want. XML tags are called nodes or elements. Each node has a type (1, which indicates an XML element, or 3, which indicates a text node), and elements may also have attributes. A node nested in a node is called a child node. This hierarchical tree structure of nodes is called the XML Document Object Model (DOM).

The main purpose of the tags is to describe the data they contain. All tags must be closed; unlike HTML, where you can leave the head tag open, in XML you must close it like this:

```
<head>a head tag</head>
```

Tags that have no data can be shortened to <head/>.

Probably one of the most important rules about tags in XML is that they must be nested correctly and that they have only one main element or the root node. All the other tags must be nested within these.

```
<?xml version= ''1.2''?>
<sprite>
  <senior programmers>
    <name>Robert Vacher</name>
    <name>Ben Salter</name>
    <name>Kay Siegart</name>
  </senior programmers>
</sprite>
```

Attributes

The tag attribute is often used in nested tags – an example of this is:

```
<book>
  <author>Alex Michael</author>
  <title>Understanding ActionScript</title>
</book>
```

The above can be rewritten in one tag using attributes.

When you have a single tag and not a block (i.e. one that doesn't have a matching closing tag), e.g. `
`, you should have a `"/` before the end, `
`; e.g. `<book author = "Alex Michael" title = "Understanding ActionScript"/>`.

We will look at the attributes tag later in this chapter. All nodes must have a beginning and ending tag, which are alphanumerically identical. That means tags are case sensitive, for instance, this is valid:

```
<authorName>Alex Michael</authorName>
```

But the following is not (note the lack of capitalization in the closing tag):

```
<authorName>Alex Michael</authorName>
```

Also note that the space after the slash and before the node name in the closing tag is optional.

Nodes which contain nodes must also be closed in the correct order. Consider a box which contains another box. You open the outer box, open the inner box then place something inside the inner box. Now if you try to close the outer box without closing the inner box, the lid won't go down because the inner box is still open. The same applies for XML, so this is valid:

```
<outerBox>
  <innerBox>toy</innerBox>
</outerBox>
```

But this is not:

```
<outerBox>
  <innerBox>toy</outerBox>
</innerBox>
```

Document Type Definitions

A Document Type Definition (DTD) is a document which sets the rules for your particular XML document. There are also standard DTDs for some established needs, such as SMIL for streaming media and SVG for vector graphics. The DTD gives detailed information about what type of data to expect in each tag. They are often used to test XML documents to make sure they conform to an expected format before an application attempts to use them.

Flash does not worry about verifying XML documents with a DTD. This means that you have to trust that the XML you are working with is in the format you expect it to be. It also means that you may have to live with the consequences if you feed your program some unexpected format.

ActionScript's XML parser is what's known as a non-validating parser. What that means is that it does not check the document it is reading against a set of rules. These rules are the DTDs or the Document Type Definitions. But because the parser is non-validating it does not acknowledge that the document does not stick to its own DTD. There are two types of DTDs. One is stored in a file separate from the XML document and this kind is not used by Flash. The second kind of DTD is part of the XML document itself and follows the document declaration. The DTD is enclosed within a single tag:

```
<!DOCTYPE rootNodeName [
  .....DTD.....
]>
```

So if the root node was named archive it would be:

```
<?xml version='2.0'?>
<!DOCTYPE archive[
  <!ELEMENT archive (book)>
  <!ELEMENT book (author, title)>
]>
```

Flash will store the DOCTYPE tag in the XML object so that it can be passed on to other systems.

Using XML Objects

You must first initiate an XML object. To do so, you need to call the XML constructor function in a new statement. This is exactly the same way as you initiate any object:

```
MyXML= new XML( );
```

Simple, what you have here is an empty XML object you use to load or read data.

Reading in XML data requires these steps:

- define the function that will be executed when the data has loaded (function myLoad in the example);
- create a new XML object (thisXML);
- set ignoreWhite to true (so Flash won't treat carriage returns and other white space in the XML file as separate nodes);
- tell Flash which function will be used when the data has loaded;
- load the data!

Here is what that looks like:

```
thisXML = new XML( );
thisXML.ignoreWhite = true;
thisXML.onLoad = myLoad;
thisXML.load( 'Sprite_page.xml' );
```

If your XML file is extremely large it may take Flash a long time to process your XML. If this takes longer than 15 seconds, the Flash Player will throw an error. You cannot change this error, or its timeout. Your options will be to reduce the size of your XML file or load and parse it in several chunks. If there are any errors in parsing a string, the object's status property will automatically indicate what kind of error occurred. Table 9.1 indicates the status property values.

Table 9.1 *Status property values*

Value	Event method
0	Successfully parsed
-2	Error in CDATA section
-3	Error with XML declaration
-4	Error with DTD
-5	Error with comment
-6	Error with element (tag)
-7	Out of memory (probably file is too big)
-8	Error with attribute
-9	Starting tag with no ending tag
-10	Ending tag with no starting tag

In the following example, the test runs through our code and detects an error. The trace command created an Output window with the value of the error. So in this example:

```
xmlStr = '<aTag>A Value';
MyXML = new XML( );
MyXML.parseXML(xmlStr);
trace(myXML.status);
```

the value of the error would be `-9`, which suggests a problem with the ending tag. The mistake is in the first line, which should look like this:

```
xmlStr = '<aTag>A Value</a>';
```

If you try to read information from another system you will find that the XML files will work fine on your local machine, but will fail when you move them onto your web server. In nearly all browsers this needs to be the server on which the HTML file sits with your Flash file. For this reason you must place your Flash file and HTML file on the same server.

The XML Tree

The first thing to do once you have created an XML object is make sure the XML is being outputted correctly by using a web browser to read in the information. Internet Explorer is a good test as it will render the XML in a tree structure as well as color the nodes and attributes. The `xmlDec1` and `docTypeDec1` properties allow you to access declarations and the DTD. Here is an example:

```
XmlStr = '<?xml version=\'1.0\'?><!DOCTYPE library
[<!Element book (#PCDATA)>]>';
Myxml = NEW xml(XMLSt);
Trace(myXML.xmlDec1);
Trace (myXML.docTypeDec1);
```

This would output:

```
<?xml version=\'1.0\'?>
<!DOCTYPE library [<!Element book (#PCDATA)>]>
```

The XMLnode Class

Nodes are hierarchical, i.e. they can contain other nodes within them (“child nodes”), and their children can contain children, just like MovieClips in Flash. XML is also associative in that two nodes which are children of the same parent node are recognized as being siblings. 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 to add the default functionality of the XML objects. Each child node is an `XMLnode` instance. The `XMLnode` class and the `XML` class have

the same methods and properties, with a few exceptions. A good test for this is running the “for...in” loop through the object for each class and outputting the value:

```
trace('the XMLnode class:');
for(i in XMLnode.prototype){
    trace(i);
}
trace('the XML class:');
for(i in XML.prototype){
    trace(i);
}
```

The result will be a long list of methods and properties in the Output window.

The child of an XML object is an XMLnode rather than another XML object. You will never try to load data into a child node. Every XML object hierarchy includes two kinds of object nodes:

- One XML node, which serves as the main container for the hierarchy.
- The other is an arbitrary number of XMLnodes, which are the children of the main container node.

The container node is an instance of the XML class. The XMLnode class does not include the loaded, status, contentType and ignoreWhite properties because the XMLnode objects never send or load data, but XML objects do. It is unlikely that you will find yourself needing to load data into a child node, but you still want to add a method or property to the class. The removeWhite() method is added to the XMLnode objects and not the XML objects because the method is recursive (calls itself); if added to the XML class instead of the XMLnode class, the recursive calls would not happen because the methods would not be defined for the XMLnode objects.

Why and When to Use XML?

XML is a very useful method of representing data and it's very much the way to go with Flash, but it has its problems, like everything. One of the biggest dangers is the verbose XML tags. The ability to create your own tags in XML means that XML documents can be self-documenting to a certain extent, but sometimes you may be drawn into abusing this feature and get into using extremely long XML tag names – for instance:

```
<book>

<BooksLabelnotInTechnicalCategorie>...</
BooksLabelnotInTechnicalCategorie>

</book>
```

The nature of this structure is fine, but the child node would of course be much more easily represented by a “reviews” node, or something much shorter. The longer your tags are, the more bytes your clients are going to have to download, and that equates directly to them spending more time waiting for your site to load, which we all know is a bad thing. For this same reason, there are many instances where XML is simply not necessary. For instance, if you have a back-end script which hands out the password for the day to various users, you might as well just send it back as a text string, instead of an XML document with one node. The tags would just be useless bytes and the new LoadVars Object contains most of the useful methods that are applicable to such a system.

For many other applications, XML is the way to go. A great example can be found at Amazon.com, who recently opened up the back-end databases (through Web Services 2) to the general public. Using the correct protocols you can send a keyword query to Amazon and it will return information on all relevant books in XML format. The document contains a list of books, each of which contains child nodes telling you everything from the author of the book to the average rating from Amazon.com reviewers. It's great and a very suitable application for XML and Flash.

Loading XML into Flash

To run this tutorial we will need an XML document to work with, so let's start small. Below I've made up a small XML document which includes just one node:

```
<author>Alex Michael</author>
```

Save this in a text file (using a Notepad or similar), as AlexMichael.xml. In Flash, create a new document 'author.fla' and save it in the same directory as the new XML file. Within Flash we need to create a new XML object into which we are going to load our external XML file.

```
tutorial_xml = new XML( );
```

By adding “_xml” to the end of my XML object's name, the ActionScript panel will automatically recognize this object as being an XML object and provide a drop-down of the appropriate methods whenever it is called.

With our new object prepared we can load in our XML:

```
tutorial_xml.load('AlexMichael.xml');
```

Flash XML methods are automatically invoked when certain events occur, such as a document finishing loading. So we do not have to write something that needs to constantly check if our data has finished loading. We define the actions we wish to take place in a function, which is invoked by the load event handler, as follows:

```
tutorial_xml.onLoad = function (finished) {
    if (finished) {
        trace(' AlexMichael.xml loaded. Contents are:
'+this.toString( ));
    }
};
```

What is happening in this code is that a function is invoked when the XML document is loaded. The function is automatically passed the `finished` argument, a Boolean value which indicates whether or not the document loaded successfully and is a built-in function of the `onLoad` event handler in Flash. If the load was successful a trace message is output which returns the following:

```
AlexMichael.xml loaded. Contents are:
<author>Alex Michael</author>
```

The `onLoad` function definition must go after the creation of the new XML object. You must place it between the object declaration and the load command. You need to create an XML object before you can assign it an `onLoad` handler. You have to assign the `onLoad` handler to the XML object before you call the load method, otherwise there won't be an `onLoad` handler.

The fact that the XML document is just one line with no trailing blank space is important. White space nodes in an XML document are parsed by the ActionScript XML parser as text nodes. This can be a problem when you are working with data because it can produce unexpected results. The `ignoreWhite` property is a property with the default setting set as `false`. When set to `true`, the text nodes that only contain white space are discarded during the parsing process. Text nodes with leading or trailing white space are unaffected. The following is a test for it.

```
myXML.ignoreWhite = Boolean (true or false)

XML.prototype.ignoreWhite = Boolean (true or false)
```

You can set the `ignoreWhite` property for individual XML objects, as in the following code:

```
myXML.ignoreWhite = true
```

As all XML objects default to `ignoreWhite` being `false`, this means that all XML objects will parse white space nodes into the XML object by default. It does not mean that any text node with white space will be discarded; what it does mean is that only text nodes that are white space only will be discarded. Most XML documents you will work with will be formatted using new-line characters, tabs. In order to make them more presentable and legible, you need to tell Flash that your XML document will contain unimportant white space. This is done by setting the `ignoreWhite` property of your XML object to `true` – for example:

```
tutorial_xml.ignoreWhite = true;
```

Here is the code for the whole example:

```
tutorial_xml = new XML( );
tutorial_xml.ignoreWhite = true;
tutorial_xml.onLoad = function(finished) {
    if (finished) {
        trace(' AlexMichael.xml loaded. Contents are:
'+this.toString( ));
    }
};
tutorial_xml.load('AlexMichael.xml');
```

Data from XML within Flash

The above example was a simple intro to loading XML into Flash; now let's look at actually using some information we've loaded. Here is a detail from the result of a query on the Amazon Web Services 2 XML server, looking for books with the word "flash" in the title. You can follow this example by visiting the Amazon website and downloading the free SDK and getting a free developer token. Amazon Web Services is a cool technology as it provides a very useful function of enabling business partners to interact with the website through standard protocols. The developer's kit contains everything you need to start using Amazon Web Services, technical documentation, sample codes – it's all there! This kit can be used to access Web Services for Amazon.com, Amazon.co.uk, Amazon.de and Amazon.co.jp. However, the documentation is in English only. To learn more about this go to www.amazon.com/xml.

This is the query string that gets the XML information below:

```
http://xml.amazon.com/onca/xml3?t=webservices-20&dev-t=[free-
developer-token-goes-here]&AsinSearch=0240519310&type=lite&f=xml
<Details url='http://www.amazon.com/exec/obidos/tg/stores/
detail/-/books/0240519310...'>
    <isbn>0240519310</isbn>
    <BookName>Understanding ActionScript: Basic Techniques for
Creatives</BookName>
    <Catalog>Book</Catalog>
    <Authors>
        <Author>Alex Michael</Author>
    </Authors>
    <ReleaseDate>20 November, 2003</ReleaseDate>
    <Manufacturer>Focal Press</Manufacturer>

    <ImageUrlSmall>http://images.amazon.com/images/P/
444444446X.01.THUMBZZZ.jpg</ImageUrlSmall>

    <ImageUrlMedium>http://images.amazon.com/images/P/
444444446X.01.MZZZZZZZ.jpg</ImageUrlMedium>
```

```

    <ImageUrlLarge>http://images.amazon.com/images/P/
4444444446X 6X.01.LZZZZZZZ.jpg</ImageUrlLarge>
    <ListPrice>£19.99</ListPrice>
    <OurPrice>£19.99</OurPrice>
</Details>

```

The XML document is a structured piece of all the information relevant to this book. You can see how I have used indentation to indicate the hierarchy of the nodes. By putting the Details node as the major node and terminating it at the end of the document, you then have a chunk of information that is self contained. It contains all the other nodes, such as ISBN, BookName, etc. as its children. Also note that the Authors node, which is a child of the Details node, has its own child node. This is because there can be more than one author who contributed to a book, so had this book had multiple authors, they would each be listed here in a separate Author node, as a child of the Authors node. The following is exactly that situation:

```

http://xml.amazon.com/onca/xml3?t=webservices-20&dev-t=[free-
developer-token-goes-here]&AsinSearch=0750659793&type=lite&f=xml

```

The result of which is:

```

<?xml version='1.0' encoding='UTF-8' ?>
_ <ProductInfo xmlns:xsi='http://www.w3.org/2001/XMLSchema-
instance' xsi:noNamespaceSchemaLocation='http://xml.
amazon.com/schemas3/dev-lite.xsd'>
_ <Request>
_ <Args>
    <Arg value='0750659793' name='AsinSearch' />
    <Arg value='us' name='locale' />
    <Arg value='[free-developer-token-goes-here]' name='
dev-t' />
    <Arg value='webservices-20' name='t' />
    <Arg value='xml' name='f' />
    <Arg value='lite' name='type' />
</Args>
</Request>
_ <Details url='http://www.amazon.com/exec/obidos/ASIN/
0750659793/webservices-20?dev-t=[free-developer-token-
goes-here]%26amp=2025%26link_code=xm2'>
    <Asin>0750659793</Asin>
    <ProductName>Marketing Through Search Optimization:
How to be found on the web</ProductName>
    <Catalog>Book</Catalog>
_ <Authors>
    <Author>Alex Michael</Author>
    <Author>Ben Salter</Author>

```

```

</Authors>
<ReleaseDate>August, 2003</ReleaseDate>
<Manufacturer>Butterworth-Heinemann</Manufacturer>

  <ImageUrlSmall>http://images.amazon.com/images/P/
0750659793.01.THUMBZZZ.jpg</ImageUrlSmall>

  <ImageUrlMedium>http://images.amazon.com/images/P/
0750659793.01.MZZZZZZZ.jpg</ImageUrlMedium>

  <ImageUrlLarge>http://images.amazon.com/images/P/
0750659793.01.LZZZZZZZ.jpg</ImageUrlLarge>
<Availability>Usually ships within 24 hours</Availability>
<ListPrice>$29.95</ListPrice>
<OurPrice>$20.97</OurPrice>
</Details>
</ProductInfo>

```

Each Author node is a “grandchild” of the Details node. Child and grandchild is the way you refer to the structure of an XML document. Most importantly for the Flash user it is the exact terminology Flash uses when manipulating XML documents. The nodes can be laid out in a family tree with each Author node as a child of the Authors node and a grandchild of the Details node, which is the “root” (top) node of the tree in this example. XML is hierarchical, so if accessing information stored deep within an XML document, such as the name of the first contributing author to a book in our example, then we must pass first through the root node, then through the appropriate child, to the grandchild and extract the information. This “drilling-down” process is often called “parsing”.

You can download the source file to this example from www.sprite.net/understanding. The fla will be UActionscrip fla. I’ve set up an XML document with the above copied into it and named it UActionscrip.xml. I’ve created a new named UActionscrip fla and into the first frame I’ve put this code:

```

// Create new XML Object and set ignoreWhite to true
UActionscrip_xml = new XML();
UActionscrip_xml.ignoreWhite = true;

// Setup load handler with function which will parse our XML
UActionscrip_xml.onLoad = function(success) {
    if (success) {
        processBook(UActionscrip_xml);
    }
};

// Load the XML file into Flash
UActionscrip_xml.load('UActionscrip.xml');

```



```
// This is the function called when the XML document is loaded

function processBook(xmlDoc_xml) {
    // xmlDoc_xml is now a reference to the XML
    // object where our information is stored

    for (a in xmlDoc_xml.firstChild.childNodes) {
        trace(a + ' ' + xmlDoc_xml.firstChild.childNodes[a]);
    }
}
```

This code is like our earlier example, except that the onLoad event handler now invokes another function which we can edit to perform operations on the XML object. Currently that function just outputs the contents of the XML node to the trace output window. This acts as a good test to see if everything is running well.

We are creating the readBook function as a way of passing on this reference instead of hard coding “UActionscrip_xml” into the function so that the function can be reused for another XML object if we wish to load in another XML file. All manipulations of the XML object will now be made within the readBook function, which accepts a reference to an XML object. The first thing we’d want to display in a book reference is the title of the book, which in this case is the bookName node of our XML.

You can keep track of where you are in the tree structure by using pointers. Flash allows code-referencing nodes within an XML document to have a virtual pointer which keeps track of where you are in the tree structure. To begin with, the pointer is pointing at the XML document or the “first child” of the document. We can move our virtual pointer to the Details node using the following code:

```
trace(xmlDoc_xml.firstChild);
```

The ‘first child’ of the Details node is the ISBN node.

```
trace(xmlDoc_xml.firstChild.firstChild);
```

The code traces the tags as well as the node’s value, which illustrates that the theoretical pointer which keeps track of where we are in an XML document is pointing to the ISBN node, not the text within it. To access the text within the tags we need to add an additional call to firstChild, as follows:

```
trace(xmlDoc_xml.firstChild.firstChild.firstChild);
```

The correct way of getting the value of the ISBN node is using the following code. Whilst the previous code output the value, we should use the nodeValue property of the XML object to get at it, so the proper way to get the value of the ISBN node is using this code.

```
trace(xmlDoc_xml.firstChild.firstChild.firstChild.nodeValue);
```

The nodeValue is an accurate representation of the string value of a node. Strings which include special characters, such as ampersands (&), using nodeValue results in the returned string including

the ampersand character. So if you wanted to access the BookName node, which is the second child of the Details node, you can use a similar approach by reaching the ISBN node and then asking for the nextSibling, which moves our virtual pointer to the next node at the same level in our tree diagram.

The first call we use is firstChild to get the Details node, another call to firstChild to get the ISBN node, then one call to nextSibling to get the BookName node.

The code looks like this:

```
trace(xmlDoc_xml.firstChild.firstChild.nextSibling.firstChild.nodeValue);
```

This traces “Understanding ActionScript: Basic Techniques for Creatives”.

As well as keeping track of the associations between nodes in terms of children, siblings, parents, Flash keeps arrays of all nodes within another node. This array is called the “childNodes” array because it lists all children for a specific node and is a way to traverse to a specific node in an XML document. If we want to view all children of the Details node as a comma separated list, we could use the following code:

```
trace(xmlDoc_xml.firstChild.childNodes);
```

To get access to the first child of the Details node, we just use standard array notation:

```
//The first element of an Array is element 0
trace(xmlDoc_xml.firstChild.childNodes[0]);
```

This function loops through each child and uses the same code to read the value of every child node:

```
function processBook(xmlDoc_xml) {
    // xmlDoc_xml is a reference to the XML object where our
    information is stored for (var n = 0; n < xmlDoc_xml.firstChild.
    childNodes.length; n++) {
        trace(xmlDoc_xml.firstChild.childNodes[n].firstChild.
        nodeValue);
    }
}
```

What we are trying to do with this code is to loop through each child of the Details node and output the value between the child node’s tags. Below is a copy of the output from the above code:

```
0240519310
Understanding ActionScript: Basic Techniques for Creatives
null
20 November, 2003
Focal Press
```

```

http://images.amazon.com/images/P/
444444446X.01.THUMBZZZ.jpg
http://images.amazon.com/images/P/
444444446X.01.MZZZZZZZ.jpg
http://images.amazon.com/images/P/
444444446X.01.LZZZZZZZ.jpg
£19.99
£19.99

```

If the output value is “null”, then the Authors node has children of its own and it is within these children that the actual names of the authors reside. So when we ask for the `nodeValue` of the Authors node, we’re told it is “null” because that node contains nothing but other nodes. In order to actually print out the names of the authors we will have to add another call to `firstChild` to drill down into the Authors node and pull the values out of the Author node children with the following code:

```

function processBook(xmlDoc_xml) {
    // xmlDoc_xml is a reference to the XML object where our
    information is stored for (var n = 0; n < xmlDoc_xml.firstChild.
    childNodes.length; n++) {
        if (xmlDoc_xml.firstChild.childNodes[n].nodeName ==
        ‘‘Authors’’) {
            trace(xmlDoc_xml.firstChild.childNodes[n].firstChild.
            firstChild.nodeValue);
        } else {
            trace(xmlDoc_xml.firstChild.childNodes[n].firstChild.
            nodeValue);
        }
    }
}

```

The above code checks through the children of the Details node, whether or not the next node to be processed is the “Authors” node, using the `nodeName` property and a conditional statement. When the Authors node is found, we move down one level to reach the name of the author of the book. This however does not take into account that there may well be more than one author involved in this book, which would mean that there would be more than one Author node within the Authors parent. One solution to handle this would be to create another loop, which iterates through the children of the Authors node, printing out each one as it goes.

You may need to store the information after you process it in some manner. It would be more efficient if you used the information as you processed it, as extensive XML documents can require large arrays or objects to store their values leading to more processing power problems. To make this more efficient a new object is created and the XML node attributes are stored within it as variables.

```

function readBook(xmlDoc_xml) {
    book = new Object();
    book.authors = new Array();
    for (var n = 0; n < xmlDoc_xml.firstChild.childNodes.length;
n++) {
        if (xmlDoc_xml.firstChild.childNodes[n].nodeName ==
''Authors'') {
            trace(xmlDoc_xml.firstChild.childNodes[n].firstChild.
firstChild.nodeValue);

book.authors.push(xmlDoc_xml.firstChild.childNodes[n].first
Child.firstChild.nodeValue);
        } else {
            trace(xmlDoc_xml.firstChild.childNodes[n].firstChild.
nodeValue);
            book[xmlDoc_xml.firstChild.childNodes[n].nodeName]
= xmlDoc_xml.firstChild.childNodes[n].firstChild.nodeValue;
        }
    }
    trace(book.ImageUrlSmall);
    trace(book.ImageUrlMedium);
    trace(book.ImageUrlLarge);
}

```

Attributes

An attribute is generally a qualifying piece of information stored within the tag of an XML node. Unlike the node's value you do not store it between the tags. An example of an attribute can be found in the first line of the XML document we've been working with all this time:

```

<Details url='' http://www.amazon.com/exec/obidos/tg/stores/
detail/-/books/0240519051...'>

```

The Details node contains an extra piece of information, the URL attribute. This occurs within the opening XML tag only, so the closing tag of the Details node is just "</Details>", it should not include the URL attribute listed in the opening tag. When you actually interface with a Web Service the XML document which is returned includes multiple Details nodes. The URL attribute can be used to distinguish quickly between one Details node and another, without having to drill down into the nodes to find out any information about them.

Attributes are stored in an object in Flash, corresponding to the level of the node they are defined in. Sticking to the same XML document we've been using to access the URL attribute we can use the following code:

```
trace(xmlDoc_xml.firstChild.attributes.url);
```

The first part is the reference to the XML object. The call to `firstChild` moves us on to the Details node, and the `attributes` call connects us to the object named “attributes” which Flash generates when processing this XML document.

Using the XMLSocket Object

ActionScript provides a built-in XMLSocket object that allows you to open a continuous connection with a server. A socket connection allows the server to publish (or “push”) information to the client as soon as that information is available. This open connection is commonly used for real-time applications such as chats. The data is sent over the socket connection as one string and should be in XML format. You can use the XML object to structure the data.

To create a socket connection, you must create a server-side application to wait for the socket connection request and send a response to the Flash movie. This type of server-side application can be written in a programming language such as Java.

You can use the ActionScript XMLSocket object’s `connect` and `send` methods to transfer XML to and from a server over a socket connection. The `connect` method establishes a socket connection with a web server port. The `send` method passes an XML object to the server specified in the socket connection.

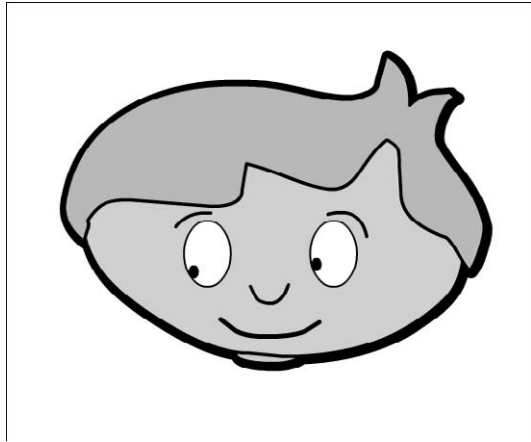
When you invoke the XMLSocket object’s `connect` method, the Flash Player opens a TCP/IP connection to the server and keeps that connection open until one of the following happens:

- The `close` method of the XMLSocket object is called.
- No more references to the XMLSocket object exist.
- The Flash Player quits.
- The connection is broken (for example, the modem disconnects).

Sending Messages To and From the Flash Player

To send messages from a Flash movie to its host environment (for example, a web browser, a Macromedia Director movie, or the stand-alone Flash Player), you can use the `fscommand` action. This action lets you extend your movie by using the capabilities of the host. For example, you could pass an `fscommand` action to a JavaScript function in an HTML page that opens a new browser window with specific properties.

To control a movie in the Flash Player from web browser scripting languages such as JavaScript, VBScript and Microsoft JScript, you can use Flash Player methods – functions that send messages from a host environment to the Flash movie. For example, you could have a link in an HTML page that sends your Flash movie to a specific frame.



WORKING WITH COMPONENTS

This Page Intentionally Left Blank

10 UI Components

The most notable advantage that the professional version has is its additional components. In the standard version there are the key components of buttons, text fields, a list box and window panes, all featured later in this chapter. The professional version includes additional elements such as an alert box, a menu component, a calendar, and many more media and data components. The data components can make your life a lot easier if you have to interface with XML, server-side languages or databases. At times you may need to use binding to bind data between components when working with XML and web services. This is outside the scope of this book.

Working with Components

Components are pre-built movie clips that you can include in your projects to add interactivity and visual effects. Macromedia provides a number of useful components, but you can also create your own and share them. It is not necessary to use components. They are provided to make your life easier. Components are movie clips with defined parameters that are set by you. The parameters correspond to variables attached to the movie clip, which affects the behavior and appearance of the movie clip. Components provide pre-built user interfaces and scripting for adding interactivity to Flash movies. With components, you can add interactivity to a Flash movie without needing to create the ActionScript that controls the interactivity. To use a component in a movie, you simply add an instance of the component symbol to the movie and select values for the component parameters.

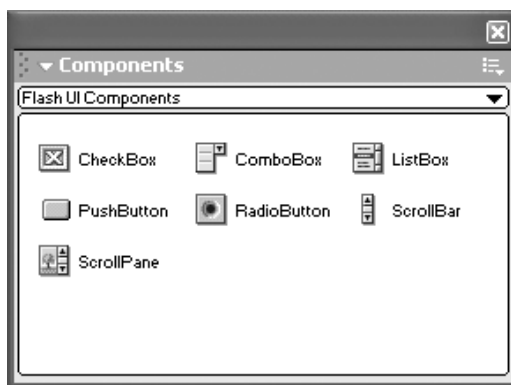


Figure 10.1 *Components from the standard version of Flash MX*

You can use components to create simple interactivity that affects only the current movie, such as buttons that navigate between frames. You can also use components to send data to an application server and receive data from a server. Towards the end of this chapter, I have put together a tutorial demonstrating the building of a navigation system using components.

The Components Panel

All components are stored in the Components panel. To display the Components panel, choose Windows>Development Panels>Components.

When you add one or more components to a Flash document, a Flash UI Components folder is added to the Library panel. This Folder contains:

- A Global Skins folder containing graphic elements.
- The component movie clip.
- Data Provider API and the class hierarchy assets within a Core Assets folder.

The library contains the components that you import and create. You create instances of a component by dragging the component icon from the library to the stage.

Skins folders contain the graphic symbols, used to display a component type in a Flash document. All components use the skins in the Global Skins folder. Components have skins folders specific to the component type. Components that use scroll bars share the skins in the FScrollBar Skins folder; ListBox components use the skins in the ComboBox Skins folder. You can edit the skins in the folders to change the appearance of components, but you cannot edit the components themselves by double-clicking the instance.

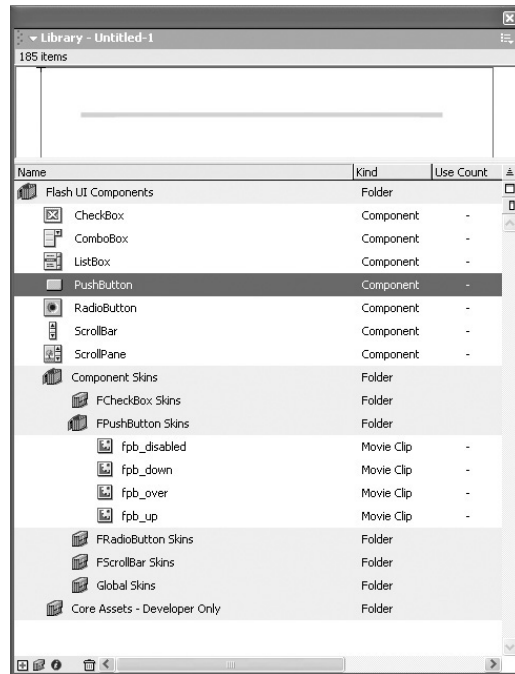


Figure 10.2 *Components in the library*

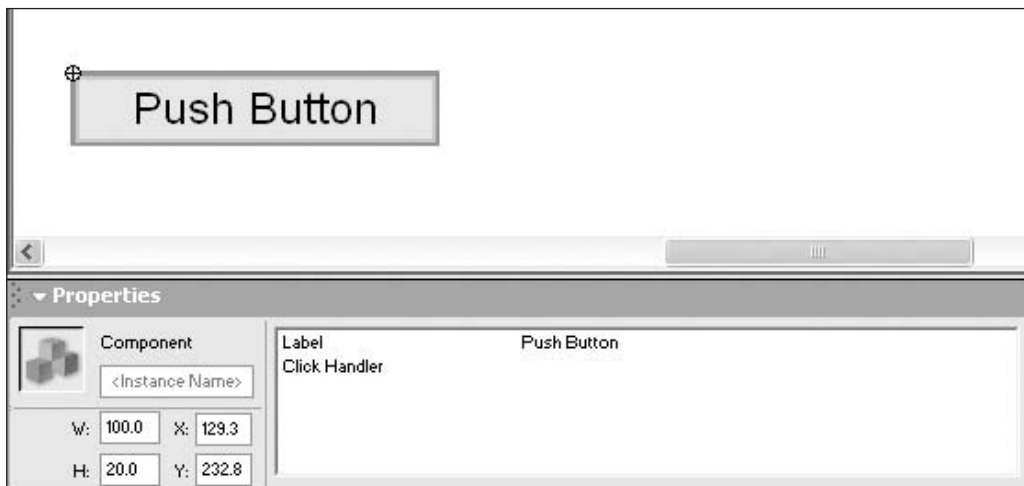


Figure 10.3 *Setting component properties*

Property Inspector and Parameters Panel

Once you have added an instance of a component to a Flash document, you use the Property Inspector to set and view information for the instance. You create an instance of a component by dragging it from the Components panel onto the stage, then you name the instance in the Property Inspector and specify the parameters for the instance.

Live Preview

The Live Preview feature lets you view components on the stage as they will appear in the published movie. Live Preview lets you see the approximate size and appearance the component will have in the published movie. Live Preview does not reflect changes you make to component property settings or to component skins. Components in Live Preview are not functional. To test, you can use the Control>Test Movie command. You can turn Live Preview on or off: choose Control>Enable Live Preview. A check mark indicates that it is enabled.

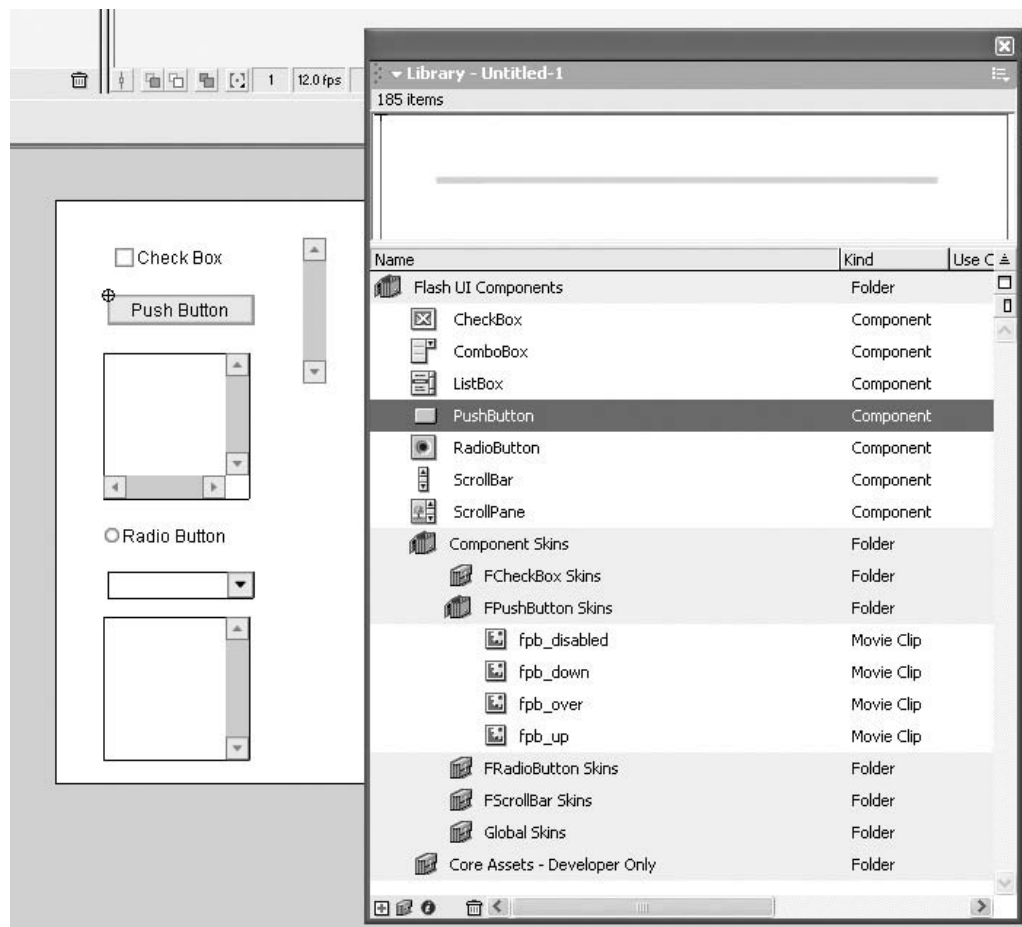


Figure 10.4 *Creating a component on the stage*

Adding Components to Flash Documents

You can add components to Flash documents using the Components panel, or using the `MovieClip.attachMovie` ActionScript method. We will focus on using the Components panel for adding components. When you add a component to a document using the Components panel, the items for the component are added to the Library panel, inside a Flash Components UI folder. To add a component to a Flash document using the Components panel:

1. Select **Windows>Development Panels>Components**.
2. Drag a component from the Components panel to the stage.
3. Select the component on the stage.
4. Choose **Window>Properties**.
5. In the Property Inspector, enter an instance name for the component instance.
6. Click the Parameters tab and specify parameters for the instance.
7. Change the size and scale of the component as desired.
8. Change the color and text formatting of a component as desired, by editing multiple properties in the global style format assigned to all Flash UI components.

Change Handlers

All components have a Change Handler. It is a function that is called when the user selects a menu item, a radio button or a check box. Specifying a function for the Change Handler parameter is optional. Using this function depends on the requirements and layout of your form or user interface, and the purpose of the component.

You can write Change Handler functions in a variety of ways. It is good practice to create a single handler function that specifies the actions for the components in your document, and then use the name of this handler function as the Change Handler parameter for the components. This makes sure that conflicting actions are not assigned, and makes it easier to update and change the code. More on Change Handlers later.

The CheckBox Component

The CheckBox component lets you add check boxes to Flash movies with simple drag and drop functionality. A check box accepts two values: true or false. You can set the following parameters for each check box instance in your Flash document using the Parameters tab on the Property Inspector or the Component Parameters panel:

- Label appears next to the check box.
- Change Handler is a function call when the value of the check box changes.
- An Initial Value specifying whether the check box is initially selected (true) or unselected (false).
- Label Placement specifies whether the label appears to the left or to the right of the check box. By default the label is displayed to the right of the check box.

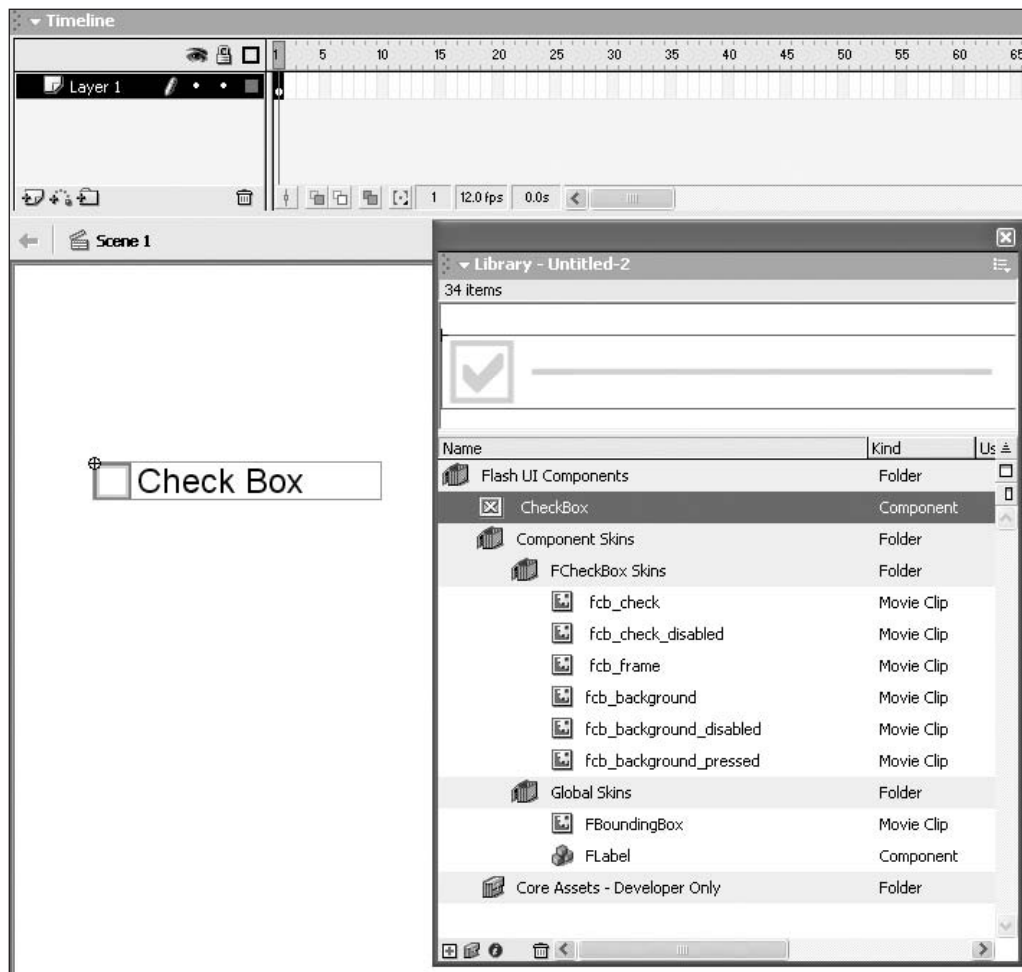


Figure 10.5 *The CheckBox component*

Figure CheckBox Properties

You can set the width, but not the height, of a CheckBox component using the Free Transform tool. The area that will respond to the mouse-click is the combined size of the check box and check box label. The CheckBox component uses the skins in the FCheckBox Skins folder and the FLabel skin in the Global Skins folder in the Component Skins folder in the library. Customizing the CheckBox component skins affects all check box instances in the Flash document.

The ComboBox Component

The ComboBox component is one of the more complicated and more powerful components. It lets you add scrollable single-selection drop-down lists to Flash movies by creating both static and editable combo boxes. A static combo box is a scrollable drop-down list that lets the user select

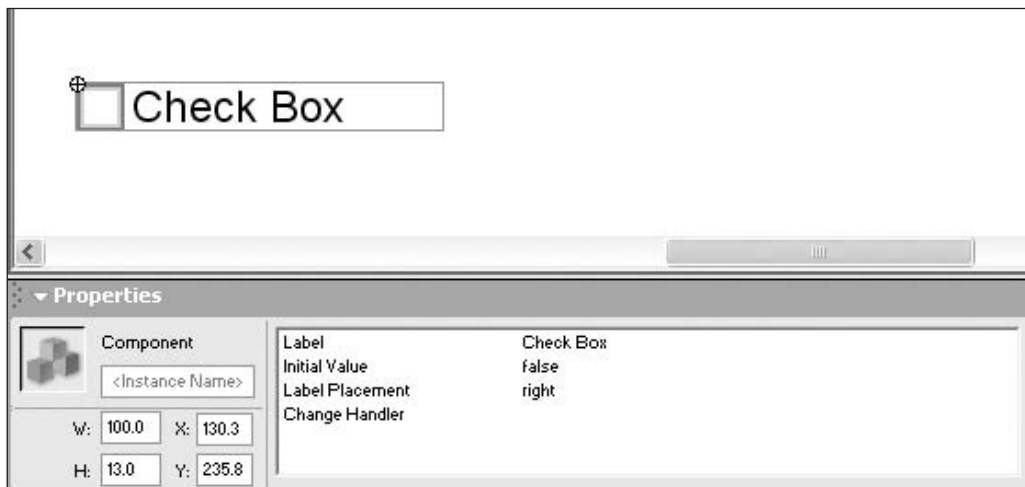


Figure 10.6 *The ComboBox component*

items in the list. An editable combo box is a scrollable drop-down list with an input text field in which a user can enter text to scroll to the matching menu item in the scroll list.

The ComboBox component uses values with the index 0 as the first item displayed. When adding, removing or replacing list items using the FComboBox methods, you may need to specify the index of the list item. The ComboBox component has the following built-in mouse and keyboard controls:

- Up and Down Arrow keys move the selection up or down one line in the scroll list.
- PageUp moves the selection up one page. The size of the page is determined by the Row Count parameter. PageDown moves the selection down one page.
- Home selects the item at the top of the list.
- End selects the item at the bottom of the list.

You can set the following parameters for each combo box instance in your Flash document using the Parameters tab on the Property Inspector or the Component Parameters panel:

- Editable tests whether the combo box is editable or static. Editable boxes allow users to enter text in a field to search for the matching items in the list.
- Change Handler is a text string specifying a function to call when a user selects an item or enters text in the input field.
- Data is an array of text strings specifying the values associated with the items (labels) in the combo box.
- Labels is an array of text strings specifying the items displayed in the combo box.
- Row Count is the number of items displayed in the combo box before the scroll bar is displayed.

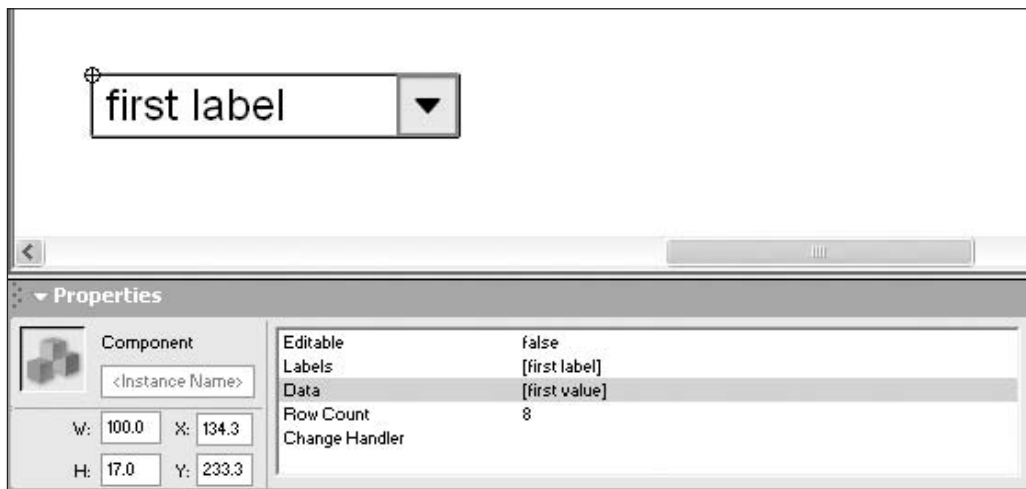


Figure 10.7 *ComboBox properties*

You can set the width but not the height of ComboBox components using the Free Transform tool. If the text of a list item is longer than the width of the combo box, the text is truncated to fit inside the box. The height of a ComboBox component is determined by the size of the font displaying the list items and the Row Count parameter, which specifies the number of items visible in the drop-down list at one time.

The ComboBox component shares the skins in the FScrollBar Skins and Global Skins folders (in the Component Skins folder in the library), with all other components that use scroll bars and bounding boxes.

The ListBox Component

A list box is very similar to the combo box, except that the list box does not require a click to see the list of items for selections. The ListBox component lets you add scrollable single- and multiple-selection list boxes to Flash movies. You add the items displayed in the ListBox using the Values dialog box that appears when you click in the labels or data parameter fields. The ListBox component uses a zero-based index, where the item with index 0 is the first item displayed. The ListBox component has the following standard built-in mouse and keyboard controls:

- The Up Arrow and Down Arrow keys move the selection up or down one position.
- PageUp moves the selection up one page. The size of the page is determined by the height of the list box instance.
- PageDown moves the selection down one page.
- Home selects the item at the top of the list.
- End selects the item at the bottom of the list.

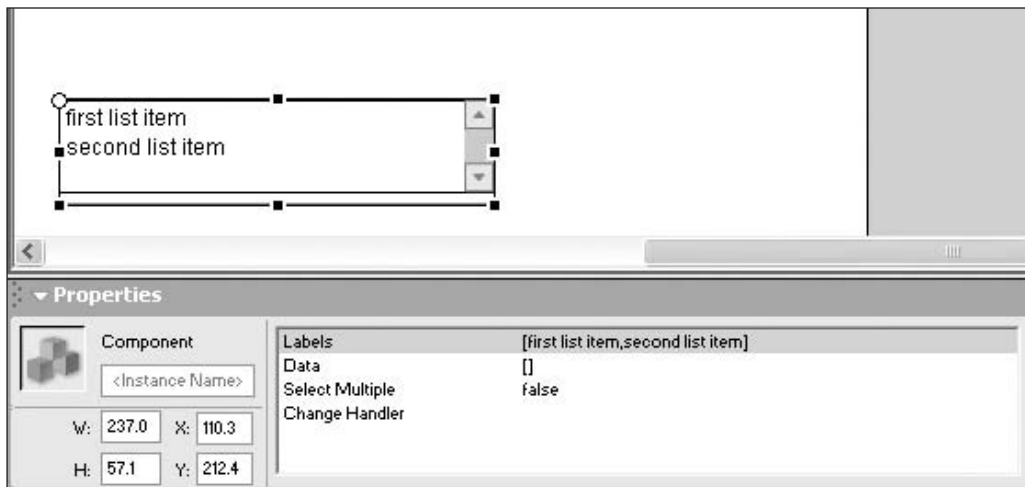


Figure 10.8 *List Box properties*

You set the following parameters for each list box instance in your Flash document using the Parameters tab on the Property Inspector or the Component Parameters panel:

- Change Handler is the name of the function that you want to call when the user selects an item in the list box. It must be defined in the same timeline as the List Box.
- Data is an array of text strings specifying the data associated with the items (labels) in the list box. It uses the value dialog box to accept input for the data array.
- Labels are an array of text strings specifying the items in the list box.
- Select Multiple specifies whether the List Box allows multiple items to be selected at one time. This is set to true to accept multiple items or false to accept only one item. The default setting is false.

You can change the width and height of list box instances using the Free Transform tool. The width of the list box instance is determined by the width of the bounding box. If the text of the list items is too long to fit inside the bounding box, the text is truncated. The height of the list box instance is automatically adjusted to display the lines of text without increasing the size of the box. The List Box component shares the skins in the FScrollBar Skins and Global Skins folders in the Component Skins folder.

The PushButton Component

The PushButton component lets you add simple push buttons to your Flash movie. The PushButton component accepts all standard mouse and keyboard interactions, and has an onClick parameter that allows you to specify a handler to execute actions. It has only two parameters that can be set during authoring. You set the following parameters for each push button instance in your Flash document using the Parameters tab on the Property Inspector or the Component Parameters panel:

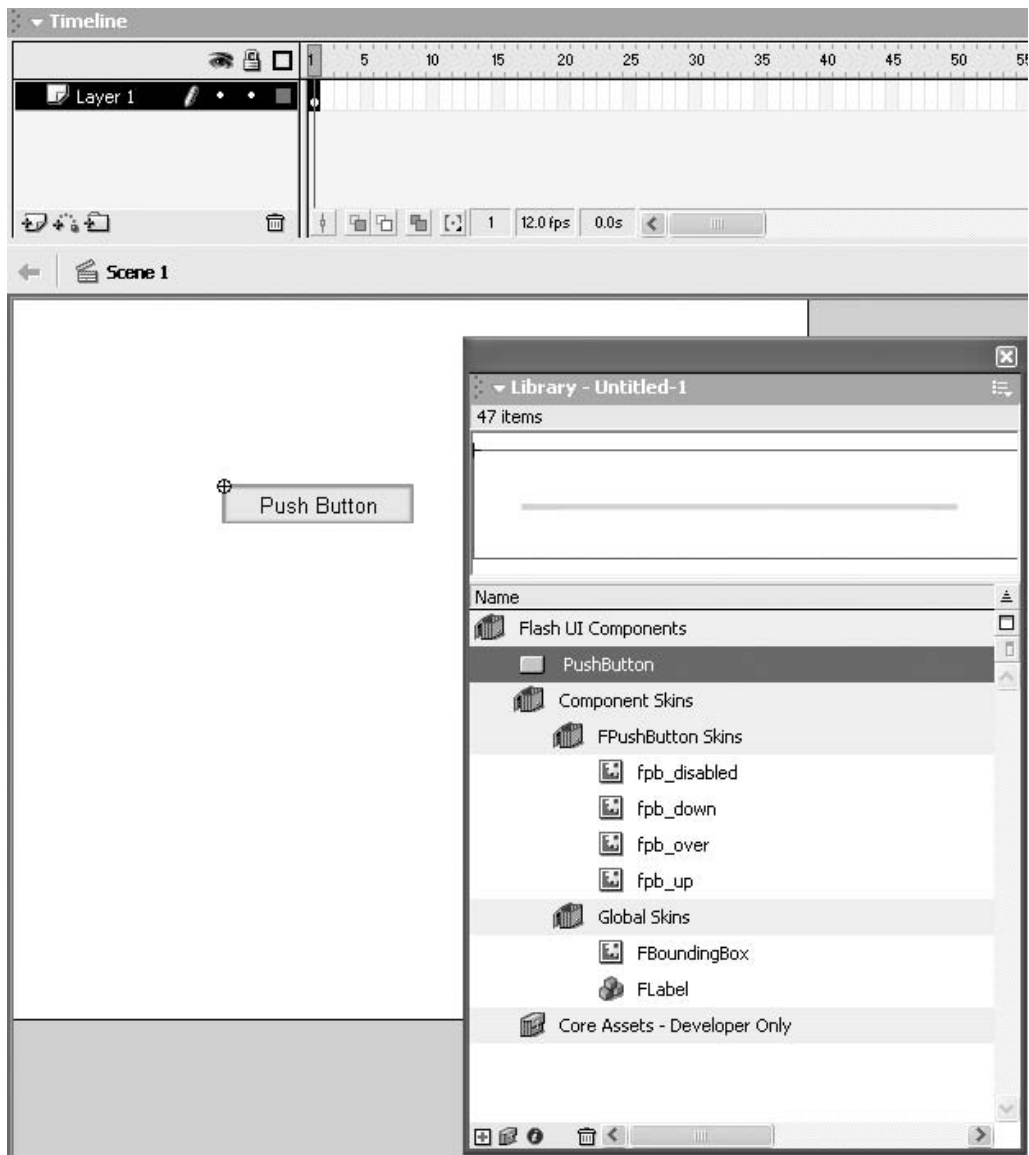


Figure 10.9 *The PushButton component*

- Click Handler is a text string specifying the function to call when a user presses and releases the push button.
- Label(s) is the text that appears on the push button.

You can size the height and width of push button instances using the Free Transform tool. If the push button is not large enough to display the label, the label text is truncated. The text is not scalable so it truncates if the size of the component is not large enough to fit the entire label. The

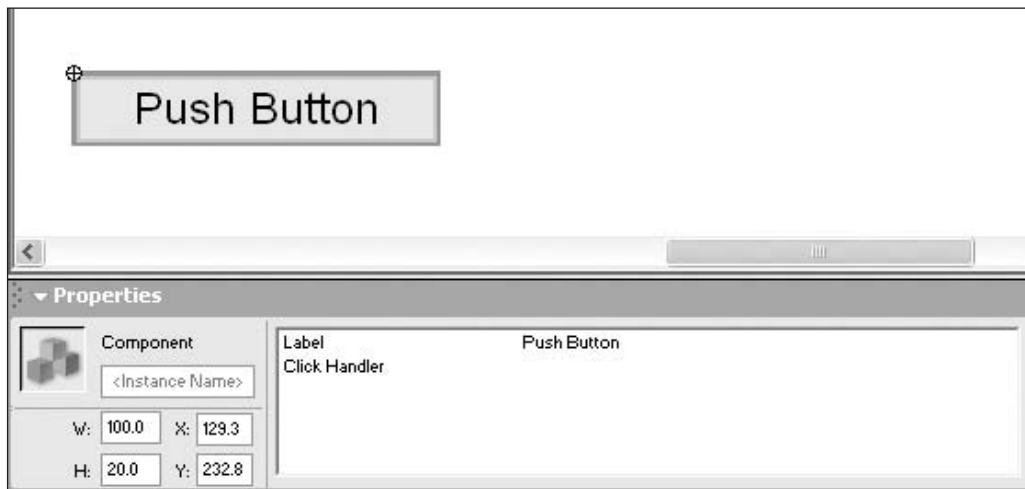


Figure 10.10 *PushButton properties*

PushButton component uses the skins in the FPushButton Skins folder and the FLabel skin in the Global Skins folder in the Component Skins folder in the library.

The RadioButton Component

The RadioButton component lets you add groups of radio buttons to your Flash document. It shares many of the components of the CheckBox and is the simplest of all the components. The groupName parameter logically groups radio button instances together and prevents more than one radio button in the same group from being selected at the same time. You can set the following parameters for each radio button instance in your Flash document using the Parameters tab on the Property Inspector or the Component Parameters panel:

- Change Handler is the name of the function that you want to execute when the user selects one of the radio buttons in a group.
- Data is the data associated with the radio button label.
- Group Name specifies the radio button as one of a group of radio buttons. If two or more RadioButtons have the same group name, then only one of them can be selected at a time.
- Initial State specifies whether the radio button is initially selected (true) or clear (false).
- Label is the name of the radio button.
- Label Placement specifies whether the label appears to the left or the right of the radio button.

You can set the width, but not the height, of RadioButton components during authoring using the Free Transform tool. The hit area of the radio button instance is the size of the radio button and radio button label area. If the radio button instance is not large enough to display the label, the label text is truncated. The RadioButton component uses the skins in the FRadioButton Skins folder and the FLabel skin in the Global Skins folder in the Component Skins folder.

The ScrollBar Component

The ScrollBar component provides drag-and-drop functionality for adding vertical and horizontal scroll bars to dynamic and input text fields. Adding scroll bars to dynamic and input text fields allows the text field to overflow without requiring that it be displayed in one go. The ScrollBar works only with dynamic and input text fields; for it to work the text field setting must be multiline no wrap or single line.

The ScrollBar component is used by the ComboBox, ListBox and ScrollPane components. Adding these components to a Flash document automatically adds the ScrollBar component to the library. If an instance of the ScrollBar component is already in the library, you can add instances of the ScrollBar component to the document by dragging them from the library. You should never add another copy of the ScrollBar component to the document by dragging it from the Components panel.

When you drag a scroll bar onto a dynamic or input text field on the stage, the scroll bar automatically snaps to the nearest side at the vertical or horizontal position where you place it. The scroll bar and the text field must be in the same timeline. The ScrollBar component cannot be used with static text fields.

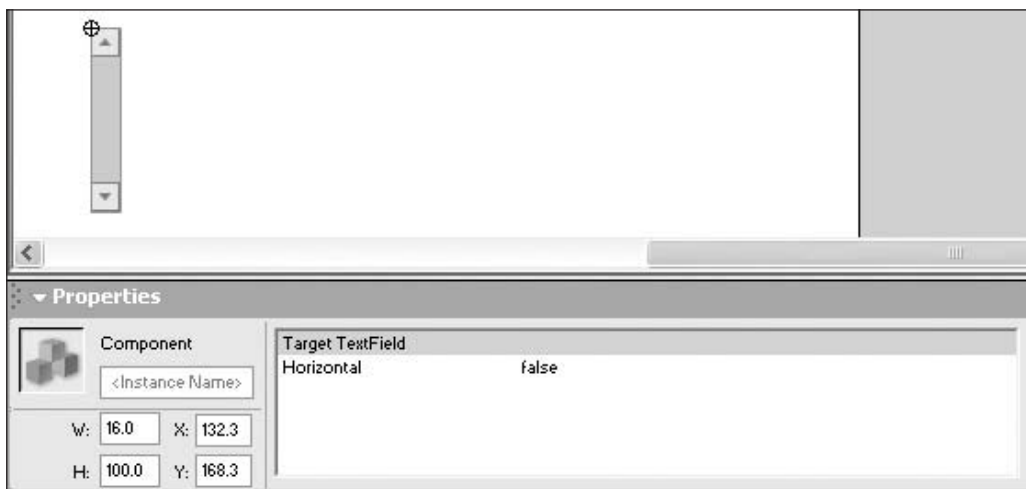


Figure 10.11 *The ScrollBar component*

Once the scroll bar is snapped to the text field, Flash enters the instance name of the text field for the target TextField parameter for the scroll bar instance in the Property Inspector. Although the scroll bar automatically snaps to the text field, it is not grouped with the text field. Therefore, if you move or delete the text field, you must also move or delete the scroll bar.

You can set the following parameters for each scroll bar instance in your Flash document using the Parameters tab on the Property Inspector or the Component Parameters panel:

- Horizontal specifies whether the scroll bar is horizontal (true) or vertical (false).

- Target Text Field is a string specifying the instance name of the text field for the scroll bar to be attached to. This parameter is automatically filled when you snap a ScrollBar to a text field on the stage. Changing or deleting this parameter disassociates the scroll bar from the text field on the stage.

Scroll bars added to text fields are automatically sized to fit the text field. If you resize the text field, the way to resize a scroll bar instance is to drag it off the text field and then back on again. You should not use the `FScrollBar.setSize` method to size scroll bars attached to text fields.

The ScrollBar component shares the skins in the FScrollBar Skins folder in the Component Skins folder in the library with all other components that use scroll bars.

The ScrollPane Component

The ScrollPane component is very similar to the ScrollBar component. The ScrollPane component lets you add window panes with vertical and horizontal scroll bars to display movie clips or JPEG files that are converted to movie clips. The ScrollPane component is useful for displaying large areas of content without taking up a lot of stage space. The ScrollPane component only displays movie clips. The best illustration of the ScrollPane is when you define a picture as a scroll component through the linkage ID. This gives you a fully scrollable picture in both horizontal and vertical directions.

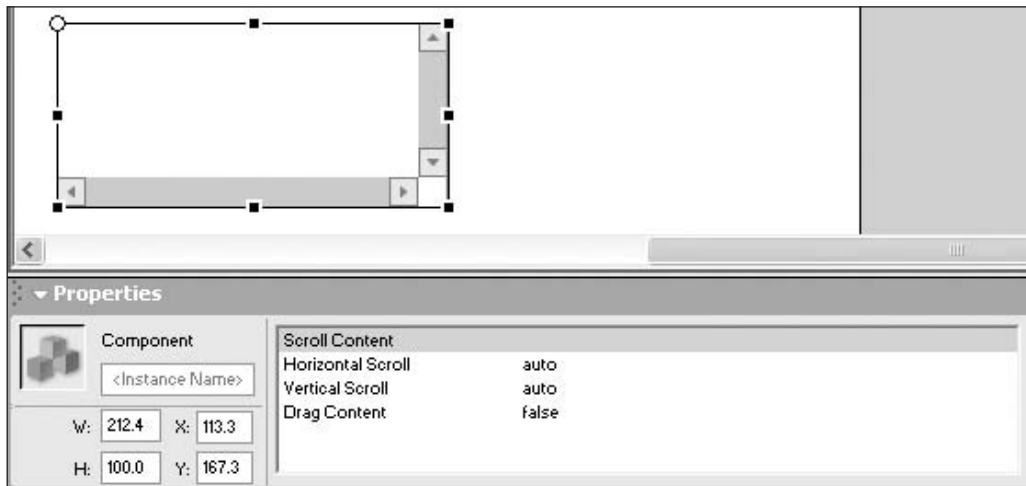


Figure 10.12 ScrollPane properties

You can set the following parameters for each scroll pane instance in your Flash document using the Parameters tab on the Property Inspector or the Component Parameters panel:

- Drag Content specifies whether a user can drag the content in the scroll pane to change the view (true), or use the scroll bars to change the view (false).
- Horizontal Scroll specifies whether a horizontal scroll bar is displayed (true), not displayed (false), or displayed only when necessary (auto).
- Scroll Content is a text string specifying the symbol linkage ID of the movie clip to be displayed in the scroll pane.
- Vertical Scroll specifies whether a vertical scroll bar is displayed (true), not displayed (false), or displayed only when necessary (auto).

You can change the width and height of scroll pane instances during authoring using the Free Transform tool. The ScrollPane component shares the skins in the FScrollBar Skins folder (in the Component Skins folder in the library) with all other components that use scroll bars.

Components Tutorial Overview

Macromedia Flash UI components can be quickly and easily developed from the selection in the library. It is also possible to customize the library components to create a new look and feel. This tutorial is designed to introduce components to beginner and intermediate Flash users and show how they can be used to easily create a simple navigation system that moves the user between frames. This tutorial will teach you how to add components to a Flash document and configure the components.

Viewing the Movie Navigation Demo

You can navigate through components in a form by doing the following:

1. Download Components.swf and Components.fla from www.sprite.net/understanding.
2. Choose File>Open and navigate to Components.swf. This is the completed movie.
3. Test each of the five sections by navigating through using the buttons and boxes in the movie. Now open Components.fla that you downloaded. This is the Flash document that generated the movie.

Adding Components

In the first step we added the components to the stage and placed them on the form. You will do the same for a radio button, a check box, a list box, a combo box and a scroll button. You will also add a push button to every section to initiate the request to move on. To add components to a document, you can either drag elements from the Components panel to the stage, or double-click them in the Components panel to place them in the center of the stage. After you add a component to a document, it appears in the document's Library panel. It is a good idea to create a new layer for the components.

1. Open the Components.fla file.
2. Choose File>Save As and save the file with a new name, such as Components2.fla.
3. Create a new layer and name it Components. You will place the components on this layer.
4. Click frame 15 in the Components layer of the timeline. Choose Insert>Blank keyframe to add a blank keyframe. Do this again for frames 30, 45, and 60.
5. Make sure the following panels are open:
 - Library panel (Window>Library).
 - Components panel (Window>Development Panels>Components).
 - Property Inspector (Window>Properties).

Adding a Combo Box

Use the ComboBox component to create a simple drop-down menu of items that can be selected by users:

1. Select frame 1 in the Components layer.
2. Drag the ComboBox component from the Components panel to the stage. Place it in the center of the screen.
3. The component appears in the Flash UI Components folder in the Library panel.

Adding a Radio Button

Use the RadioButton component to create a box with a value of either true or false:

1. Select frame 15 in the Components layer.
2. Drag the RadioButton component from the Components panel to the stage. Place it in the center of the screen and add another three RadioButtons below the first one.
3. The component appears in the Flash UI Components folder in the Library panel.

Adding a Check Box

Use the CheckBox component to create a box with a value of either true or false:

1. Select frame 30 in the Components layer.
2. Drag the CheckBox component from the Components panel to the stage. Place it in the center of the screen and add another three CheckBoxes below the first one.
3. The component appears in the Flash UI Components folder in the Library panel.

Adding a List Box

Use the ListBox component to create a box with Labels and Data:

Labels	Data
ComboBox	ComboBoxFrm
RadioButton	RadioButtonFrm
CheckBox	CheckBoxFrm
ScrollPane	ScrollPaneFrm

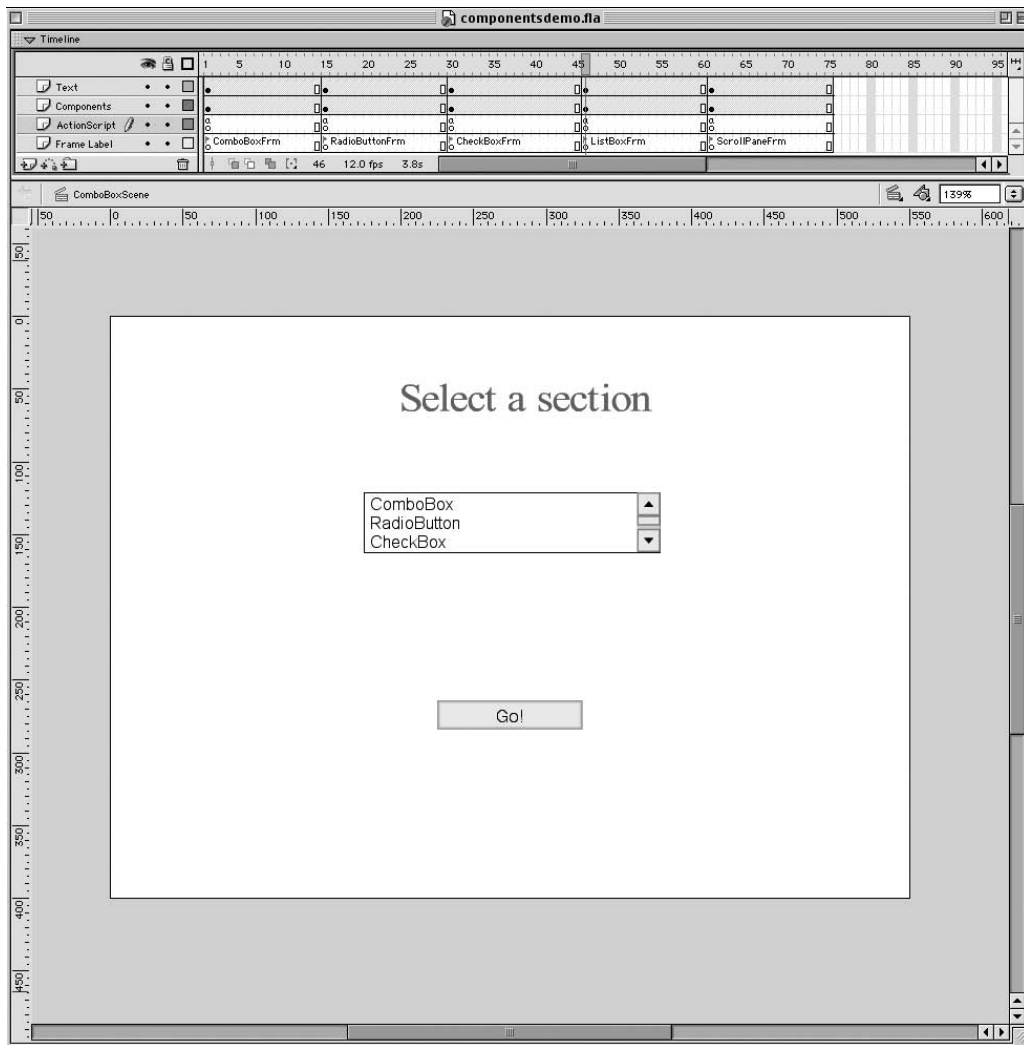


Figure 10.13 *The list box on stage*

1. Select frame 45 in the Components layer.
2. Drag the ListBox component from the Components panel to the stage. Place it in the center of the screen and add the information in the properties window for the labels and the data associated with those labels.
3. The component appears in the Flash UI Components folder in the Library panel.

Adding a Scroll Pane

Use the Scroll Pane component to create a window for scrolling around graphics:

1. Select frame 60 in the Components layer.

2. Drag the ScrollPane component from the Components panel to the stage. Place it in the center of the screen and add the information in the properties window for the contents of the pane, ScrollPane Contents. ScrollPane Contents is a movie that sits in the library.
3. The component appears in the Flash UI Components folder in the Library panel.

Adding Push Buttons

Use the PushButton component to create push buttons on frames 15, 30, 45 and 60. The buttons will be used to submit the information on the “form” in each section:

1. Drag the PushButton component from the Components panel to the stage. Place it at the bottom of the screen.
2. The component appears in the Flash UI Components folder in the Library panel.
3. After you have dragged a component from the Components panel to the stage, you select additional instances of it from the Library panel.

Testing Your Movie

To see what the components look like, run your movie in the Flash Player:

1. Select Control>Test Movie.
2. The movie is run in the Flash Player.
3. Select and deselect the check box to be sure it works.
4. When you are finished, select File>Close to close the movie in the player.

Configuring the Components

The next step is to configure the components so that all the clicks become meaningful actions. You set the parameters for a component using the Parameters tab of the Property Inspector or the Components Parameters panel. Advanced users can use API methods and properties for each component to set the parameters, size, appearance and other properties of the component.

Configuring the Combo Box

Select frame 1 on the Components layer, then select the RadioButton component on the stage. Its parameters are displayed in the Property Inspector.

1. Type FrameSelector in the Instance Name text box.
2. Type ComboBox in the Label text box. Do the same for CheckBox, ListBox and ScrollPane.
3. Type ComboBoxFrm in the Value text box. Do the same for CheckBoxfrm, ListBoxfrm and ScrollPanefrm.
4. In the Initial Value parameter pop-up menu, select true for all the buttons. This specifies whether the initial state of the RadioButton component is selected (true) or unselected (false).

5. The Label Placement should be set to right alignment. The label will be displayed to the right of the check box. Do not put in anything for the Change Handler parameter. When you finish, the Property Inspector should look like Figure 10.14.

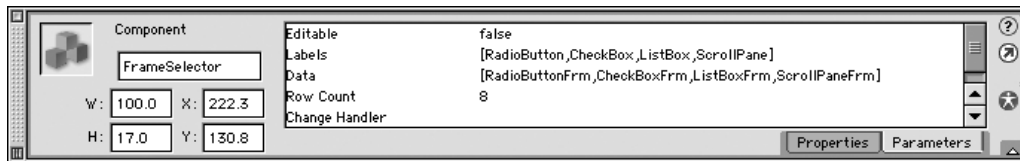


Figure 10.14 *The ComboBox Property Inspector*

Configuring the Radio Buttons

Select the RadioButton component on the stage. Its parameters are displayed in the Property Inspector.

1. Type RadioButtonFrm in the Instance Name text box.
2. Make sure the Editable parameter is set to false. This prevents users from typing in their own text.
3. The Labels parameter displays a list of values users can select. Click the Labels field, then click the magnifying glass to open the Values pop-up window. Click the Plus (+) button to enter a new value.
4. Click in the default value field, then type RadioButton for the first value.
5. Click the Plus (+) button to enter the next value. Click in the default value field, then type CheckBox for the next value and ListBox for the next, and ScrollPane for the next. There is no need to enter a Change Handler parameter name as the handler is attached to the button. When you are finished, the Property Inspector should look like Figure 10.15.

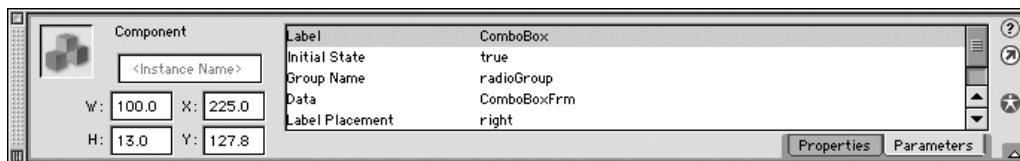


Figure 10.15 *The RadioButton Property Inspector*

Configuring the Check Box

In this example the first selected item will be used to guide the user to the next section. Select frame 1 on the Components layer, then select the CheckBox component on the stage. Its parameters are displayed in the Property Inspector.

1. Type CheckBoxFrm in the Instance Name text box.
2. Type CheckBox in the Instance Name text box and in the Label text box.

3. In the Initial Value parameter pop-up menu, select true.
4. Leave the default value set to right alignment. The label will be displayed to the right of the check box. Do not alter the Change Handler parameter. When you finish, the Property Inspector should look like Figure 10.16.

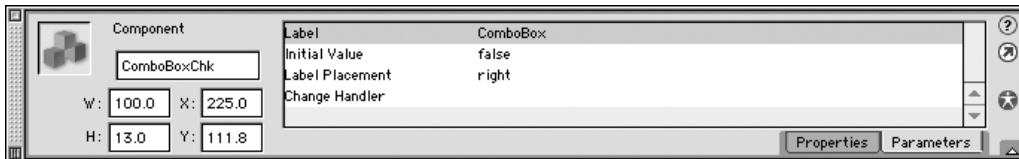


Figure 10.16 *The CheckBox Property Inspector*

Configuring the List Box

The ListBox component lets you add scrollable single- and multiple-selection list boxes to Flash movies. Select the ListBox component on the stage. Its parameters are displayed in the Property Inspector.

1. Type ListBox in the Instance Name text box.
2. Type ComboBox in the Label text box. Do the same for CheckBox, ListBox and ScrollPane.
3. Type ComboBoxFrm in the Value text box. Do the same for CheckBoxfrm, ListBoxfrm and ScrollPanefrm. The select multiple should be set to false.
4. Select frame 45 on the Components layer, then select the ListBox component on the stage. Its parameters are displayed in the Property Inspector.
5. Select true for all the buttons. The Label Placement should be set to right alignment. Do not put in anything for the Change Handler parameter. When you finish, the Property Inspector should look like Figure 10.17.

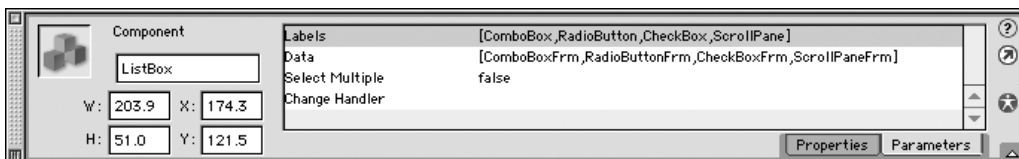


Figure 10.17 *The ListBox Property Inspector*

Configuring the Scroll Pane

Use the ScrollPane component to create a window for scrolling around graphics.

1. Select the ScrollPane component on the stage. Its parameters are displayed in the Property Inspector.

2. Type ScrollPane Contents in the Scroll Contents text box.
3. In the Drag Content box set false to change the view using the scroll bars. Set horizontal and vertical scrolling to auto. Do not put in anything for the Change Handler parameter. When you finish, the Property Inspector should look like Figure 10.18.

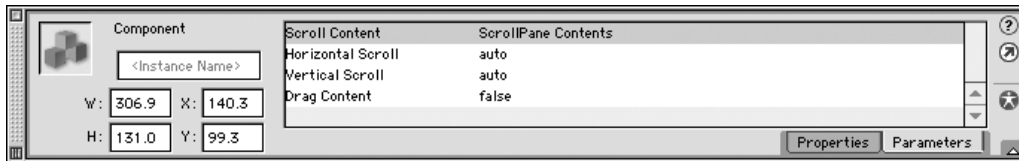


Figure 10.18 *The ScrollPane Property Inspector*

Configuring the Push Buttons

Select the PushButton component in frame 1. The component's parameters are displayed in the Property Inspector.

1. Type Go! in the Property Inspector Label text box. This text is displayed in the body of the button.
2. Type jumpToFrame1 for the Click Handler name. You will write the Handler later. When you are finished, the Property Inspector should look like Figure 10.19.



Figure 10.19 *The PushButton Property Inspector*

3. Select the PushButton component in frame 15. Do the same as above for the label and type jumpToFrame2 for the Click Handler name.
4. Select the PushButton component in frame 30. Do the same as above for the label and type jumpToFrame3 for the Click Handler name.
5. Select the PushButton component in frame 45. Do the same as above for the label and type jumpToFrame4 for the Click Handler name.
6. Select the PushButton component in frame 60. Do the same as above for the label and type jumpToFrame5 for the Click Handler name.

Writing ActionScript for the Entire Movie

ActionScript for components is placed in keyframes. The Click Handler parameter specifies what happens when the PushButton component is activated. Our values are set to jumpToFrame*,

which means that when the user clicks on a push button, it is activated. We will begin by creating a function for `jumpToFrame1`. (A fully working version of this tutorial exists on the website ([\(www.sprite.net/understanding\)](http://www.sprite.net/understanding))(`compDemo.fla`).)

1. Create a new layer and name it `ActionScript`. This will be used for ActionScript that should run throughout the movie.
2. If the Actions panel is not open, choose `Window>Development Panels>Actions`.
3. Switch to Expert Mode by pressing `Control+Shift+E` (Windows) or `Command+Shift+E` (Macintosh), or by clicking the control in the upper right corner (a triangle with a check mark above it) and selecting `Expert Mode` from the pop-up menu.
4. First, enter the Submit button handler in frame 1. Enter the following code in the Actions panel:

```
stop( );
// This function is called as the 'Click Handler' function of
// the 'Go!' PushButton.
function jumpToFrame1( ) {
    // Use the getValue( ) function of the ComboBox to get
    // the value of the currently selected item.
    var next_frame = _root.FrameSelector.getValue( );
    gotoAndPlay(next_frame);
}
```

5. Enter the Submit button handler in frame 15. Enter the following code in the Actions panel:

```
stop( );
// This function is called as the 'Click Handler' function of
// the 'Go!' PushButton.
function jumpToFrame2( ) {
    // Use the getValue( ) function to get the value of the
    // currently selected radio button.
    var next_frame = _root.RadioGroup.getValue( );
    gotoAndPlay(next_frame);
}
```

6. Enter the Submit button handler in frame 30. Enter the following code in the Actions panel:

```
stop( );
// This function is called as the 'Click Handler' function of
// the 'Go!' PushButton.
function jumpToFrame3( ) {
```

```
// Using the getValue( ) function of each CheckBox, this
// function checks each checkbox in turn and jumps to its
// associated frame.
if (_root.ComboBoxChk.getValue( )) {
    gotoAndPlay( ''ComboBoxFrm'' );
} else if (_root.RadioButtonChk.getValue( )) {
    gotoAndPlay( ''RadioButtonFrm'' );
} else if (_root.ListBoxChk.getValue( )) {
    gotoAndPlay( ''ListBoxFrm'' );
} else if (_root.ScrollPaneChk.getValue( )) {
    gotoAndPlay( ''ScrollPaneFrm'' );
}
}
```

7. Enter the Submit button handler in frame 45. Enter the following code in the Actions panel:

```
stop( );
// This function is called as the ''Click Handler'' function of
// the 'Go!' PushButton.
function jumpToFrame3( ) {
    // Using the getValue( ) function of each CheckBox, this
    // function checks each checkbox in turn and jumps to its
    // associated frame.
    if (_root.ComboBoxChk.getValue( )) {
        gotoAndPlay( ''ComboBoxFrm'' );
    } else if (_root.RadioButtonChk.getValue( )) {
        gotoAndPlay( ''RadioButtonFrm'' );
    } else if (_root.ListBoxChk.getValue( )) {
        gotoAndPlay( ''ListBoxFrm'' );
    } else if (_root.ScrollPaneChk.getValue( )) {
        gotoAndPlay( ''ScrollPaneFrm'' );
    }
}
}
```

8. Enter the Submit button handler in frame 60. Enter the following code in the Actions panel:

```
stop( );
// This function is called as the ''Click Handler'' function of
// the ''Go!'' PushButton.
function jumpToFrame5( ) {
```

```
// This uses the same method as on Frame 1 to  
// use a combo box to navigate the movie.  
var next_frame = _root.FrameSelector2.getValue( );  
gotoAndPlay(next_frame);  
}
```

9. Now test your movie.

The next chapter applies some of your components skills to a quiz game.

11 Trivia Quiz Game Built Using Components

Introduction

This game demonstrates the use of Flash MX components to build a multiple choice trivia quiz. The game loads the questions from an external file, with each question having three possible answers. Radio buttons, push buttons and scroll bars are used to construct the interface for the game. XML is used to import the questions.

Building the Game – Game Structure and Title Page

First we need to create the background for the quiz and the structure for the way the game will work. First, name the bottom layer “Background” and place the “Background” asset from the library onto the stage. This forms the backdrop for the game. You can download a copy of the final FLA from www.sprite.net/understanding.

Next, create a layer and name it “Labels”. Create keyframes on frames 1, 15 and 30 and label them “startQuiz”, “displayQuestion” and “endQuiz” respectively. The quiz will work by starting on the “startQuiz” frame and then looping on the “displayQuestion” frame, each time displaying a different question. When all the questions have been answered, the “endQuiz” frame is played which displays their score and prompts the player to start the quiz again.

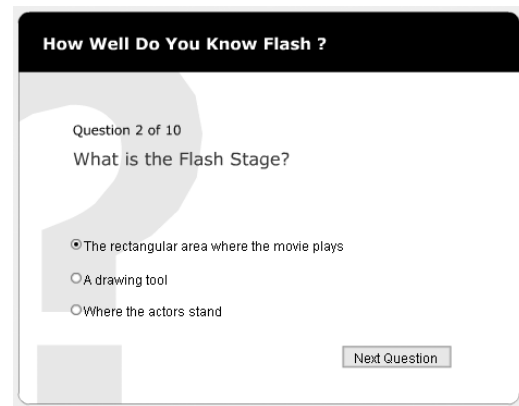


Figure 11.1 *Flash MX trivia quiz*

To display the instructions, create a third layer and name it “Instructions”. On the first frame of this layer, create a text box, set its type to “Dynamic Text” and select “Multiline” from the “Line Type” drop-down box. This text field will display the instructions for the quiz and will demonstrate the use of the Scrollbar Flash MX component. Paste the text from instructions.txt into the text field and select Text>Scrollable on the menu bar. You can now resize the text field to any size you want by selecting it with the text tool and dragging the bottom right corner of the text field.

Now we have the instructions, we need to attach a scroll bar to make it scrollable. From the Components panel, drag a ScrollBar onto the stage and align it next to the text field. To link the text field and ScrollBar give the text field the name “instructions” on the Properties panel. Next, select the ScrollBar and set the “Target TextField” value to “instructions”. Compile and view the movie to see it in action. See demo1.fla for the complete example.

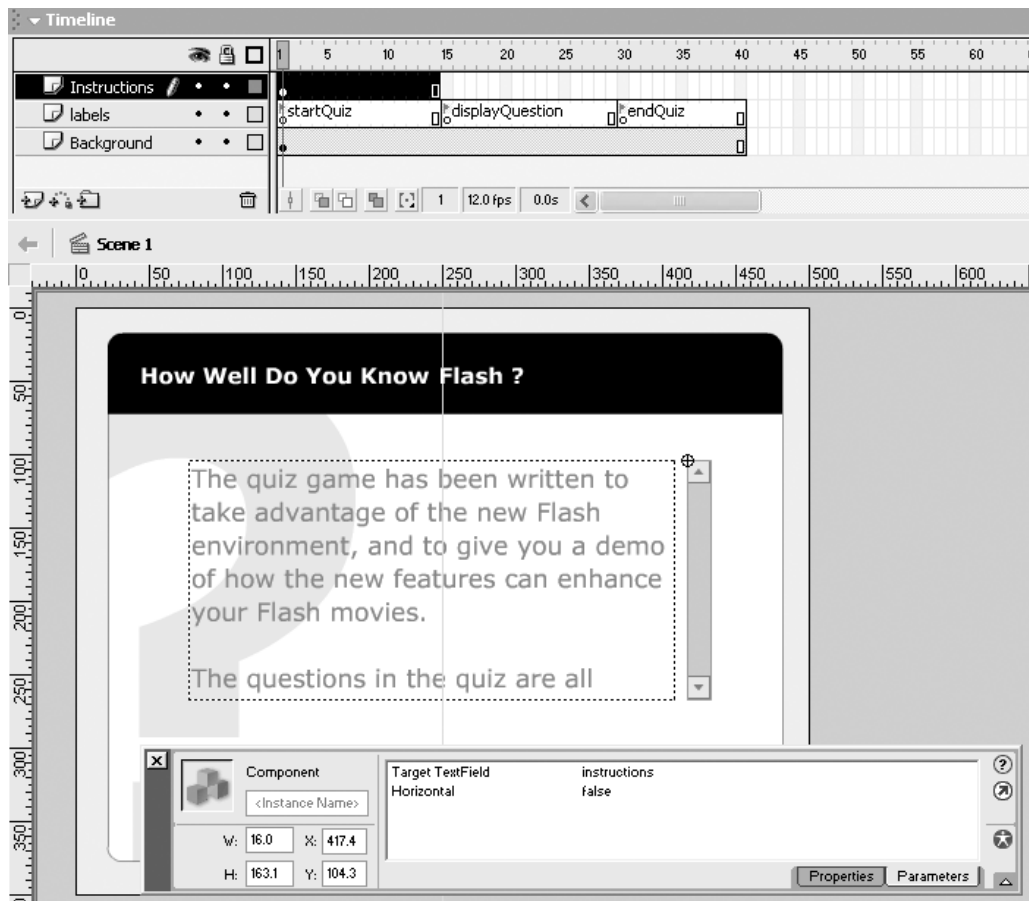


Figure 11.2 *Adding a scrolling text field*

Building the Game – Loading the Questions

Now that we have the structure of our quiz we need to add some questions. We do this by loading them in from a file. The quiz uses XML to simplify the loading and management of the quiz questions.

The questions XML file

XML (Extensible Markup Language) is a great way to interchange structured data in Internet applications. We have used XML in this quiz to format the questions so they can be read by Flash in a standard format. In XML, as with HTML, you can use tags to mark up, or specify, a body of text. In XML, you define tags that identify a piece of data, which allows the same XML document to be used and reused in different environments.

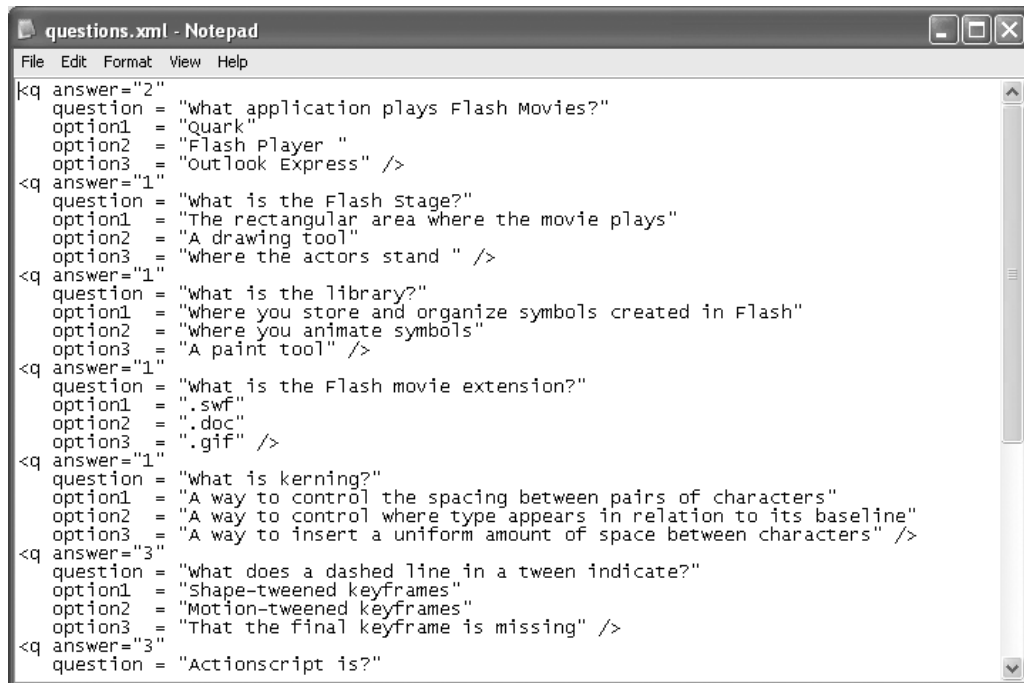


Figure 11.3 *The questions XML file*

Below is an example of how an XML question is set out:

```
<q answer="2"
  question = "What application plays Flash Movies?"
  option1 = "Quark"
  option2 = "Flash Player "
  option3 = "Outlook Express" />

<q answer="1"
  question = "What is the Flash Stage?" ...
```


You will see the format is similar to HTML, with the question defined as name = value pairs in a `<q>` tag. The first question sets the answer for the question as “2”, the question text as “What application plays Flash Movies?” and the three possible answers “Quark”, “Flash Player” and “Outlook Express”; as you can see the answer is the second option. You can download a copy of questions.xml from www.sprite.net/understanding for the complete set of questions.

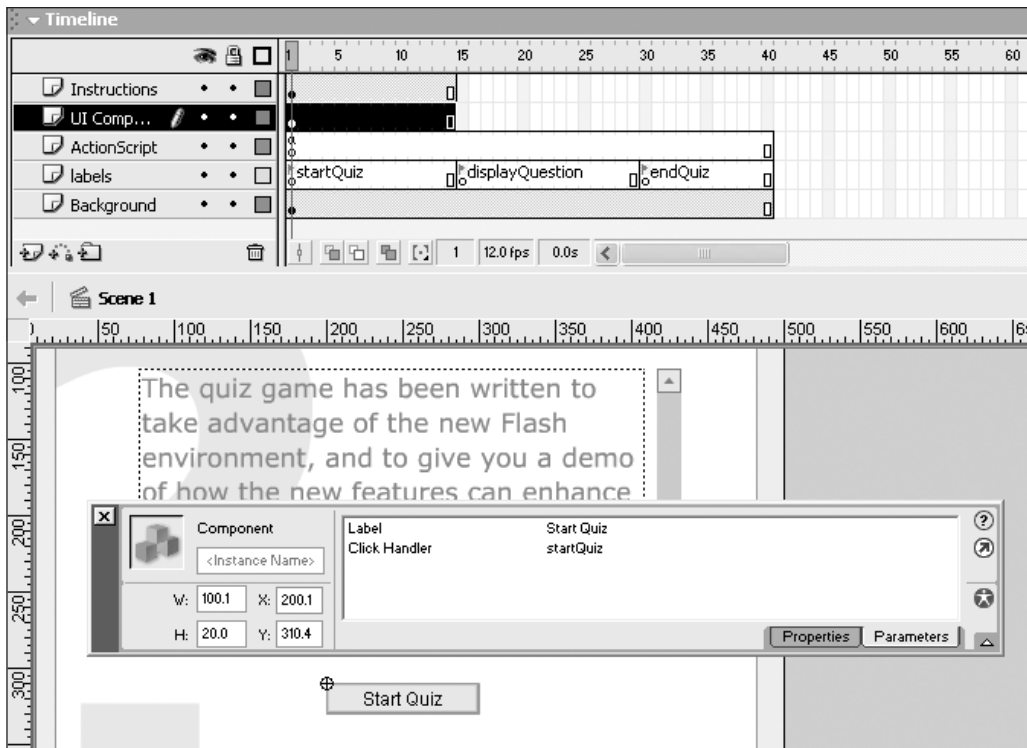


Figure 11.4 Adding the start quiz button

Before loading the questions we need to add a button to let the player start the quiz. Create a new layer named “UI Components” and add a new PushButton by dragging one onto the stage from the Components panel. Select the PushButton and, on the Properties panel, set the “Label” value to “Start Quiz” and the “Click Handler” to “startQuiz”. The “Click Handler” value is the name of the ActionScript function that will be run when the button is clicked.

Next, create a new layer and call it “ActionScript”. Create a keyframe on the first frame and insert the following code to the first frame action:

```

stop( );
/*-----
    startQuiz( )
    This is the click handler for the ''Start Quiz'' button.
    The function creates a new XML object to load and
    manage the questions for the quiz. The appropriate
    set of questions is loaded into the XML object and
    the quiz started.
    -----*/
function startQuiz( ) {
    // Create a new XML object to store the questions
    // for the quiz
    questionsXML = new XML( );
    questionsXML.ignoreWhite = true;
    questionsXML.onLoad = loadedQuestions;

    // the current question number
    _root.questionNumber = 0;
    // the number of correct answers
    root.correctQuestions = 0;

    // Load in the question set
    questionsXML.load(''questions.xml'');
}

/*-----
    loadedQuestions( )
    This is the onLoad handler for the questionsXML XML
    object. This is called when an XML document is loaded
    into the object. The formats the XML document into an
    array of XML question objects.
    -----*/
function loadedQuestions( ) {
    // Format XML document into an array of XML objects.
    _root.questions = this.childNodes;

    // Display the first question.
    gotoAndPlay(''displayQuestion'');
}

```

startQuiz() ClickHandler function

The startQuiz() function is called when the player clicks the “Start Quiz” button. It begins by creating a new XML object to store the questions in, then initializes the current question number and score, and loads the questions using the load() method of the XML object. The loadedQuestions() function is called when the question set has been loaded from the file; this function then displays the first question by moving to the “displayQuestion” frame.

Building the Game – Displaying the Questions

Now that we’ve got some questions, we need to display them and get the player’s answer. Create a new layer and name it “Question Display”. Create a keyframe at frame 15 and on it create two text fields. The first will display the question number and the second will display the current question. Set the “Var” value on the properties panel to “questionIndicator” for the first text field and “questionText” for the second, larger, text field.

To display the options for the answer we’ll use three radio buttons. These are ideal as they only allow one option to be selected at a time. Place three RadioButtons onto the “UI Components” layer at the “displayQuestion” frame. Name the RadioButtons “option1”, “option2” and “option3”. Select each RadioButton and set the Data value to “1”, for the first, “2” for the second and “3” for the third. Make sure that the groupName for each RadioButton is set to “radioGroup” and that the Label is blank for each one.

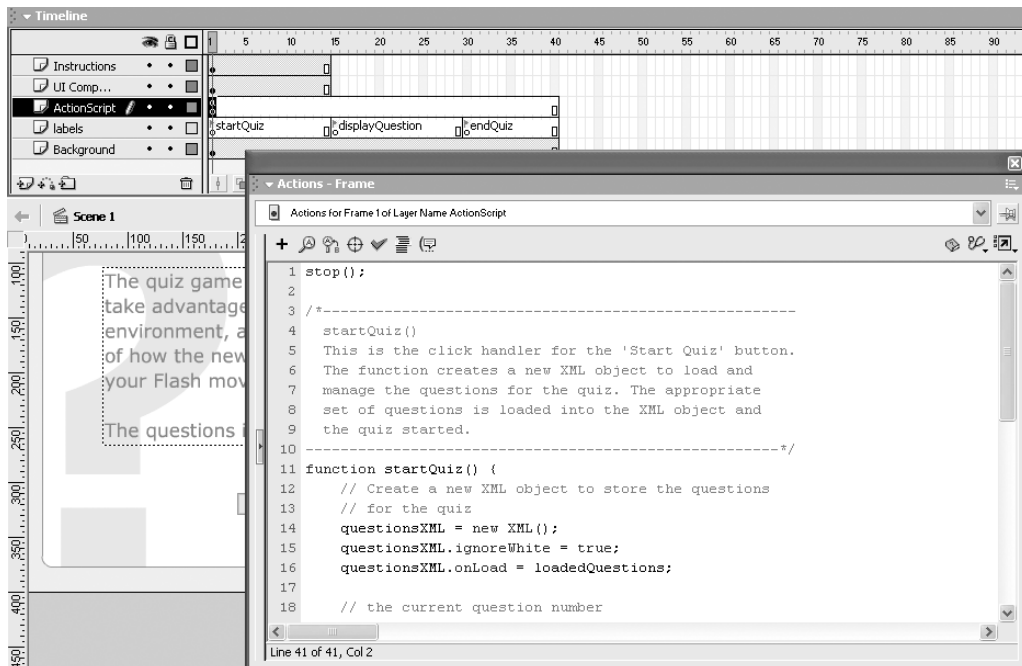


Figure 11.5a XML object to store the questions

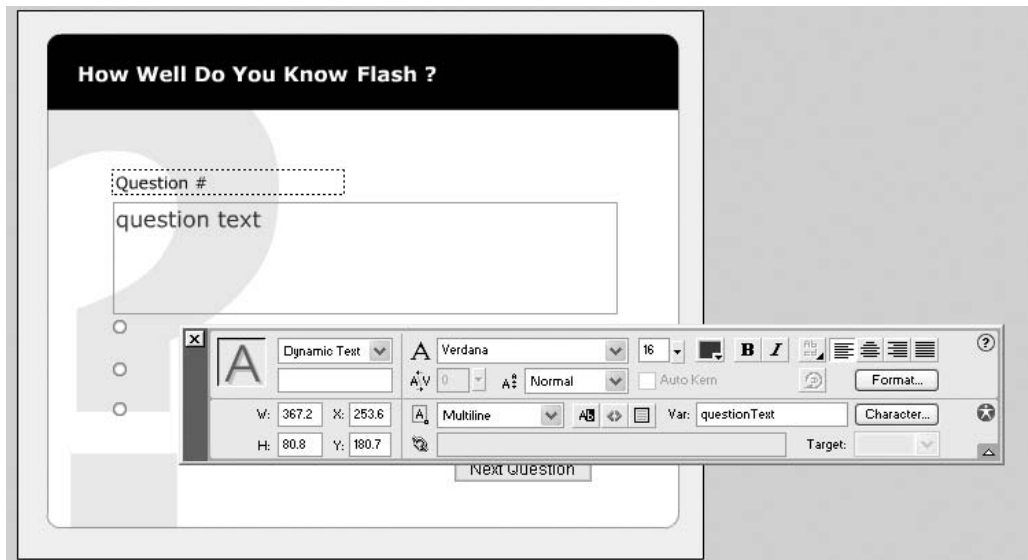


Figure 11.5b *Adding question display components*

Finally, add another `PushButton` to the “UI Components”, giving it the name “`nextQuestionButton`” and the value “Next Question”. This button will be used to proceed to the question when an answer is selected.

To display the question, add the following code on the `ActionScript` layer at frame 15:

```
// Disable the “Next Question” button until
// an answer is selected.
nextQuestionButton.setEnabled(false);

// Update the question number indicator
questionIndicator = “Question ” + (questionNumber + 1) +
    “ of ” + questions.length;

// Get the current question XML object
currentQuestionXML = questions[questionNumber];

// Update the UI component labels
questionText = currentQuestionXML.attributes.question;
option1.setLabel(currentQuestionXML.attributes.option1);
option2.setLabel(currentQuestionXML.attributes.option2);
option3.setLabel(currentQuestionXML.attributes.option3);
stop();
```

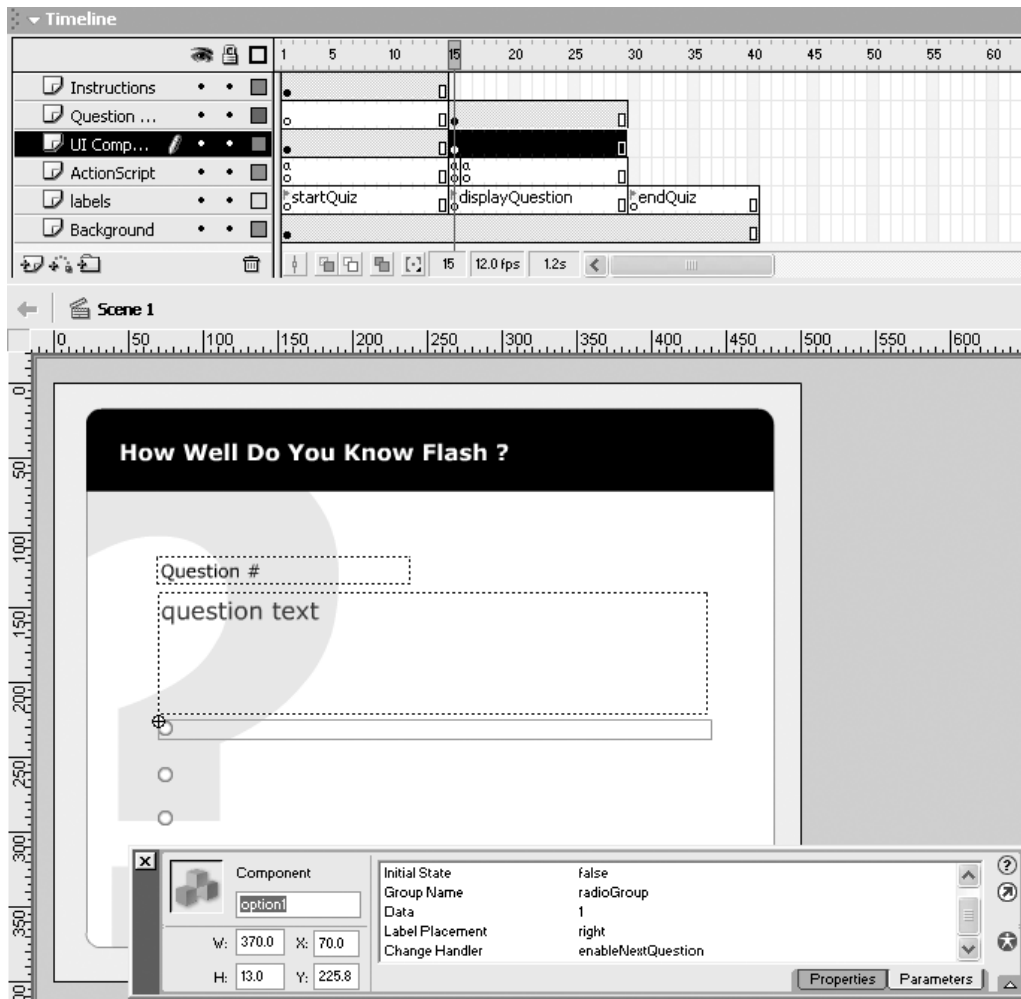


Figure 11.6 Adding comments to layers

This code first disables the “Next Question” button until an answer is selected. It then updates the first text field to say “Question 1 of 10” depending on the current question. The question data is then retrieved, the question displayed and labels of the radio buttons updated to display the possible answers. To change the label of a RadioButton, you use the `setLabel()` function giving it a new label, e.g.

```
// This would change the label of option1 to 'Hello World'
option1.setLabel('Hello World');
```

Now that we’ve got a way to display the question and select an answer, we need to add the code to take us onto the next question. First we’ll need to enable the button when they’ve selected an

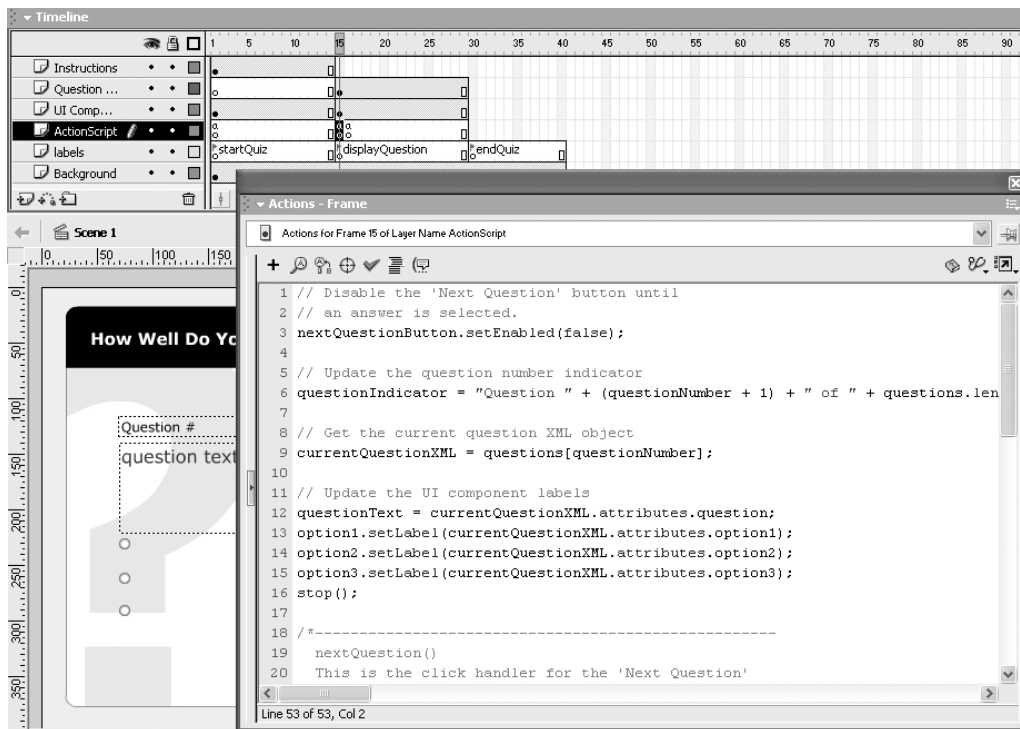


Figure 11.7 Code to display question

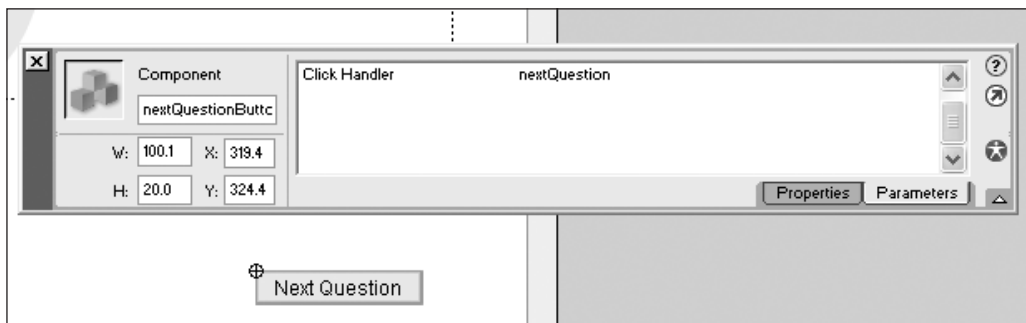


Figure 11.8 Adding the “Next Question” button

answer. To do this we attach a “Change Handler” to each radio button, so that when a radio button is selected the “Next Question” button is enabled. To do this, open the Properties panel and set the “Change Handler” value of each radio button to “enableNextQuestion”. Next we need to add the ActionScript function to enable the button; we add the following code to the frame 15 frame action:

```

/*-----
enableNextQuestion( )
This is the changeHandler for each of the radio
buttons. This enables the 'Next Question' button when
they select an answer.
-----*/
function enableNextQuestion( ) {
    nextQuestionButton.setEnabled(true);
}

```

Running the Flash movie now, you will see the button is enabled when you select a radio button. We now need to add functionality to the “Next Question” button. Select the button and set the “Click Handler” to “nextQuestion”, then add the following code to frame 15:

```

/*-----
nextQuestion( )
This is the click handler for the 'Next Question'
button.
-----*/
function nextQuestion( ) {
    questionNumber++;

    // If their answer is correct, add to their score
    if (radioGroup.getValue( ) ==
        currentQuestionXML.attributes.answer) {
        correctQuestions++;
    }

    // If they've answered all the questions goto the
    // end of the quiz
    if (questionNumber == _root.questions.length) {
        gotoAndPlay('endQuiz');
    }

    // Reset the radio buttons to be all deselected
    option1.setState(false);
    option2.setState(false);
    option3.setState(false);
    // Display the next question
    play( );
}

```

This code gets the value of the currently selected radio button and compares it to the answer of the current question. If they match, then the number of correct answers is incremented by one. The next check is to see if they have answered all the questions. If so, then the movie jumps to the “endQuiz” frame. Otherwise, the radio buttons are all deselected by calling the `setState()` function for each and the frame repeats by playing the next frame. Frame 16 simply has the frame action:

```
gotoAndPlay('displayQuestion');
```

See `demo3.fla` for a finished version of this section. The code for this section can be found in `code2.txt`.

Building the Game – The Finishing Touches

To finish the game we need to add an end of game screen that displays the player’s score and allows them to play again. Create a new layer named “End Text” and create a keyframe on it at frame 30. On this keyframe, create a dynamic text field to display the end of game message and set its “Var” value to “welldone”. Add the following code to frame 30, on the ActionScript layer:

```
correct = (100/questions.length) * correctQuestions;

// Display a different message depending on how many
// questions they got right.
if (correct <= 20) {
    welldone = 'Oh no, that was a woeful effort, only '
} else if (correct <= 50) {
    welldone = 'Not bad, but you've still got a long way
               to go to prove your Flash knowledge. '
} else if (correct <= 70) {
    welldone = 'Nearly there, try again and you might
               just crack it. '
} else {

    welldone = 'You've done it, you're a super brain!!
               (now get away from that computer and get
               some fresh air) '
}
```

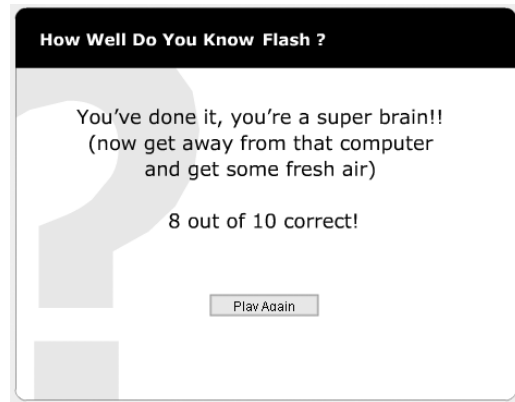


Figure 11.9 End of game screen


```
// Display how many questions they got correct.
welldone += '\n\n' + correctQuestions + ' out of ' +
           questions.length + ' correct!'
stop();
```



Figure 11.10 Adding the end of game message

Adding the end of game code

This code works out what percentage of the questions the player got correct and updates the text field appropriately. Finally, we need to give the player the ability to try again. For this, add a PushButton from the Components panel onto the “UI Components” layer on frame 30. On the Properties panel, set the “Label” to “Play Again” and the “Click Handler” to “restartQuiz”. We now need to add the restartQuiz() function; we add the following code to the frame action:

```
/*-----
restartQuiz()
This is called when the "Play Again" button is clicked.
-----*/
```

```
function restartQuiz( ) {  
    gotoAndPlay( ''startQuiz'' );  
}
```

Now simply compile and run the movie. The finished movie can be found as `quiz.fla` and the code for this final section in `code3.txt` (see www.sprite.net/understanding).

External XML data does not have to come from a static XML file. You can just as well call up a CGI, JSP or other kind of script or program that generates dynamic XML data. You can use this technique to import data from databases.

This Page Intentionally Left Blank



GAME BUILDING

This Page Intentionally Left Blank

12 Goose Game

Introduction to Design and Programming of an Application

The main objective of this chapter is to put everything you have learned into building a game with the support of a step-by-step breakdown and explanations of all the code. The game is available from www.sprite.net/understanding and is made up of a simple syntax:

```
duplicateMovieClip( )  
  
else  
  
for  
  
if  
  
with  
  
startDrag( )  
  
onClipEvent( )
```

The Game Loop

A game can be decomposed into five processes that continually run into a loop. It is a basic formula that almost every game has to have. The game loop is a series of processes for getting input from and displaying output to the user and updating the game. The five basic processes are:

1. **The Introduction** – The start screen has an animation sequence that shows off some aspect of the game's story or background and a start button to play the game or to change various parameters that affect the game in some way. The parameters include sound volume, graphic options, difficulty and starting level.
2. **Player Input** – These actions will take the player's input from whatever device is being used, and store it in a way that the game can process in order to make changes to the game internals.
3. **Updating Game** – These routines are the real guts of the game. Everything from moving the player's character using their input, to the actions of the enemies and deciding whether the player has won or lost the game is determined here.

4. **Displaying the Screen** – You can do this in two ways: either draw everything to the screen at one time and present this as a refresh or, as is more commonly done, set everything up to be drawn and then draw the screen afterwards; this could be because your character has moved onto a different level. You need to find the smallest of changes to refresh your screen, as redrawing the screen can take longer than most processes in a game. Determining what is necessary to be drawn can take a relatively long time with all the checks your game could have, so it is best to find ways of testing the processor insensitivity of graphics before you commit to large sequences.
5. **Ending the Game** – Once the game is over there is normally an ending sequence. An ending sequence can invite the player to play again or even to type out their details to load onto a high scoreboard that sits on the web.

Understanding what elements go into creating a game loop can definitely make starting your first game easier. The following methodology for identifying the assets and scripts is key to making this process reusable for future projects. Before you start programming anything, sit down and plan it out – the end result will be much better.

Methodology for Identifying Code and Assets for Game Development

The initial approach is to make a textual description of the game or application you are building. This stage involves producing a description of the game which can be broken up into its basic objects. When writing the description try to include as much detail as possible about aspects of the game such as attributes and actions associated with things in the game. The more detail you include the easier it is to decompose into a complete structured design.

Identify Assets by Extracting Nouns

For example, “you control a **cross hair** that tracks **geese** moving between the right-hand side of the screen to the left. You can fire **shots** to destroy the **geese**. The birds can hide behind clouds. At a certain time the game is over.” The nouns identify the top level assets (or objects) in the design and in Flash these directly translate to MovieClip assets. The example in Table 12.1 defines the player and goose assets.

Table 12.1 *The assets*

Geese	Cross hair	Clouds	Shots
Goose Repeats with geese	Projectile Repeats with cross hair	No repeats	No repeats
Birds Repeats with geese			

Once the nouns have been identified it is important to remove any duplicated or misleading assets. For example, the Projectile asset is potentially misleading and is simply an instance of either the

missile or cross hair assets. Similarly, birds refer to the geese asset and goose is simply another way of describing the geese asset. This ensures that each asset is unique and not simply a duplicate or specific type of another asset.

Table 12.2 *The properties associated with each asset*

Goose	Properties
	Pilot
	Damage
	Score
	Ammo
	Type
	Color
Bullet	Properties
	Velocity
Cross hair	Properties
	Velocity

Identifying the Properties Associated with Each Asset

Each asset in the game design contains a number of properties that define specific features about the instance of that asset. For example, the goose will have a damage level, score and ammo, while the goose is of a certain type and color.

In Flash, each movie clip asset has its own in-built properties, these include `_x`, `_y`, `_width` and `_height`, which are useful for controlling the size and position of the asset. Global properties that are available to all assets in the game can also be defined by accessing the `_root` object.

Asset Preparation: Identifying the Situations and Functions Associated with Each Asset

Once the separate assets and their properties have been identified the situations that the assets can be in and functions they perform need to be obtained. An asset is typically a graphical image and as such may appear differently in different circumstances; we will call these **Situations**. For example, a goose could be in a variety of situations, be it moving left, right, firing or exploding. These situations may require new images and animations, all of which must be taken into account when designing each asset.

As well as the graphical situations, there are also the actions that need to be performed which we don't see; these we will call **Functions**. A function could be used to calculate the score, player damage or winning status. For example, a function could be used to determine whether the player has been destroyed.

To begin identifying the possible situations that an asset can be in, the verbs need to be extracted from the textual description. For example, “the goose is **moved** using the direction keys and the cross hair **fires** a missile when the space bar is pressed. The goose craft **descends** and **attacks** by dropping bombs at random.” From this, the top-level situations move and fire can be extracted for the goose and its assets and the other situations inferred.

In fact, many graphical animations can actually be achieved using simple transformations of a single image; for example, a goose flying away may only require a single image reduced in size over time. This reduces the development time and suitable situations should be identified after this stage.

Table 12.3 *The properties, situations and functions associated with each asset*

Goose	Properties
	Type
	Color
	Damage
	Score
	Ammo
	Situations
	Descend
	Attack
	Move(direction)
Cross hair	Explode
	Functions
	DetectHit
	EndGame
	IncrementScore
	Properties
	Velocity
	Situations
	Move
	Explode
Bullet	Functions
	Properties
	Velocity
	Situations
	Move
	Explode
	Functions

Introduction

Using the above asset plan, the following walkthrough will demonstrate how to create a simple action game and make effective use of animation and sound to further enhance the experience. Geese fly across the screen from right to left and your aim is to shoot as many geese as you can before your time runs out. You control a target with the mouse and click to fire. The game is available for download from www.sprite.net/understanding.



Figure 12.1 *Goose Attack!*

Building the Game – Opening Screen and Assets

The opening screen features a flying goose and a prompt to start the game. The background consists of a Sun and clouds moving across the movie. Add the “Sun” movie clip from the library to the stage on a new layer named “Sun”. Align the Sun to the top left corner of the stage.

The clouds are created by using a motion tween to animate the clouds moving. Open the “Clouds” asset from the library to see this animation in action. You will notice that the last frame is the same as the first, so that the movie loops smoothly and continuously. Add the “Clouds” movie to the stage onto a new layer called “Clouds”. Using the Properties panel, align the movie clip to 270, 156. Also using the Properties panel, set the tint color to the

background color and set the amount to 40%; this is used to blend the clouds into the background. Using the tint effect is useful for altering the color of a movie clip without having to change the movie clip itself.

Create a new layer named “New Game” and place the “Title” movie clip from the library onto the stage. This displays the title of the game. The goose is a four-frame animation. Open “Goose Fly” to see each keyframe of the animation. Place an instance of the “Goose Fly” movie on the “New Game” layer below the title. Next, place the “Instructions” movie clip below the goose and the “Start Game” button below that.

To add some background sound, create a new layer called “Sound” and, from the Properties panel, select “ATMO.WAV” from the Sound drop-down box. Set the loop to 1000 times so that the sound loops for the duration of the game. Finally, create a new layer called “ActionScript” and on the first frame add a `stop()` action to pause the movie on the opening screen.

Create a keyframe on frame 3 and give it the label “startGame”. Add the following code to the action of the start game button:

```
on(release) {  
    gotoAndPlay('startGame');  
}
```

This button will now start the game.

Building the Game – Generating the Geese

The game works by using the `duplicateMovieClip` function to create new instances of the geese. Every frame, each goose is moved from right to left until they reach the other side of the screen or are shot down. They are then deleted and a new goose created on the right.

Create a new layer named “Source Assets”; this will hold our geese to be duplicated. Create three instances of the “Goose Fly” movie clip on this layer and name them “goose_far”, “goose_med” and “goose_near”. The three geese are used to add the appearance of depth to the game. Resize the geese so that “goose_far” is the smallest and “goose_near” is the largest. Set the tint color for “goose_far” and “goose_med” to the background color and set the amounts to 40% and 25% respectively. By tinting the geese it helps give the impression that they are further away. To make the geese move at different speeds, we need to give each a specific speed that it should move at, i.e. how far left it moves every frame. To set this, add the following code to the object action of “goose_far”:

```
onClipEvent(load) {  
    speed = 5;  
}
```



Figure 12.2 Adding a tint to the clouds

This will set the furthest goose to move 5 pixels left every frame. To “goose_med” add the code:

```
onClipEvent (load) {
    speed = 7;
}
```

and to “goose_near” add the code:

```
onClipEvent (load) {
    speed = 13;
}
```

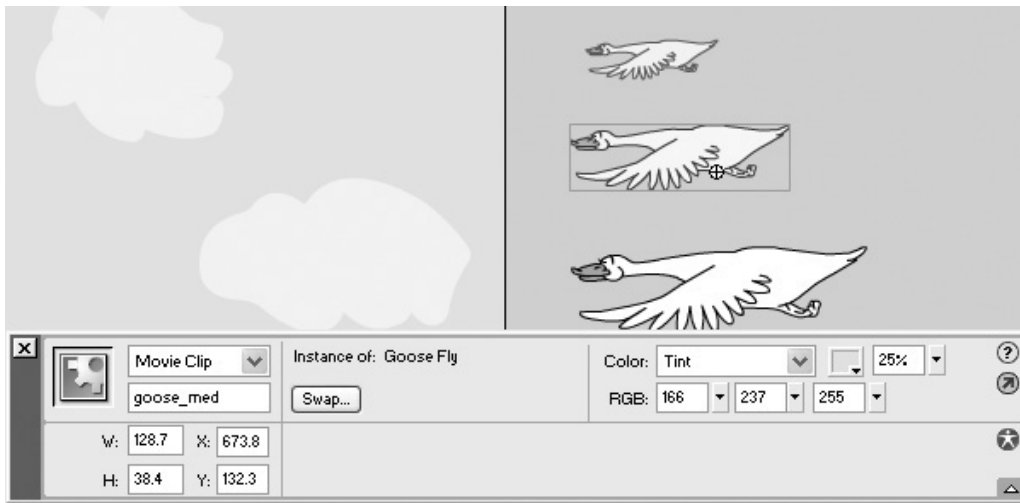


Figure 12.3 Adding the geese

These settings will be used to make the furthest geese move slower than the nearer ones.

Next we need an indicator to tell the player how much time has elapsed and their score. To do this, create a layer called “Score” and create a keyframe on frame 3. Type two labels, “Time” and “Score”, and create two dynamic text fields next to the labels. On the Properties panel, give the time text field the “Var” value “timeleft” and the score text field the value “score”. The “Var” value becomes an ActionScript variable that can be used to update the text field value by simply setting the variable to another string.

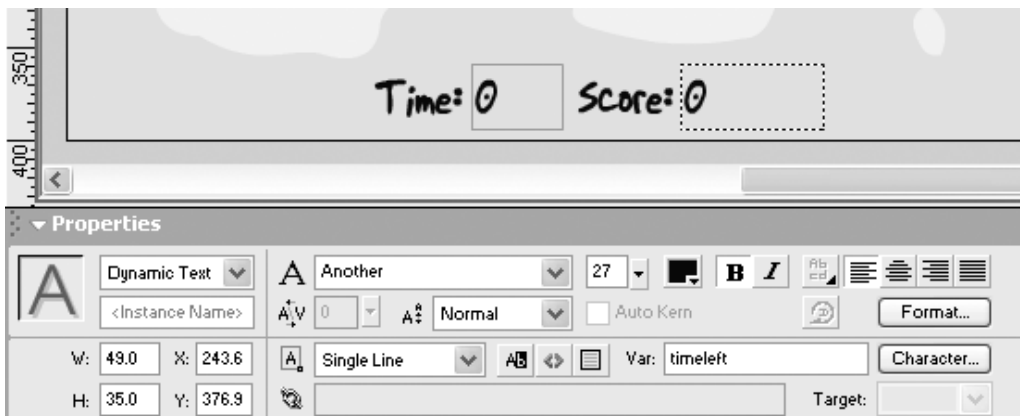


Figure 12.4 Adding the time and score indicators

To start the game we need to set some initial settings. On frame 3, add the following code to the “ActionScript” layer:

```
var game_score = 0;
var movie_width = 550;
var movie_height = 400;
var fps = 24;
var time = 30;
timeleft = time;
```

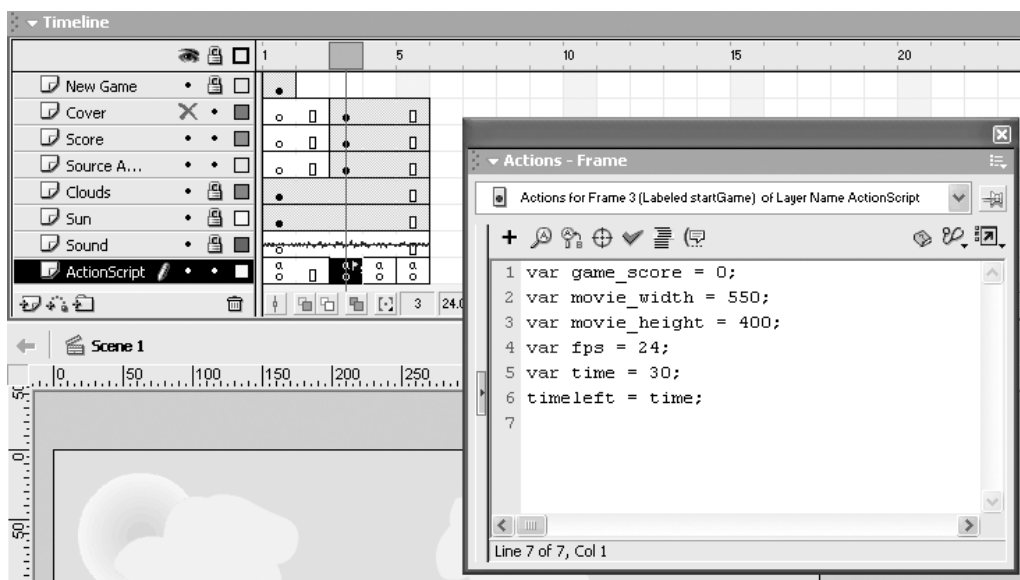


Figure 12.5 Setting startup variables

This code sets the starting score to zero, it defines the dimensions of the Flash movie, the speed that the movie runs at and the time limit for the game. The final line updates the “timeleft” text field to indicate how much time remains.

Next we need to add the ActionScript to generate and move the geese; to do this, create a keyframe on frame 4 of the “ActionScript” layer and add the following code:

```
for (i = 0; i < 6; i++) {
    // If the goose is not present, create a new one
    // at a random and position off screen
    if (!_root[“goose” + i].flying) {
        if (i < 2) {
```

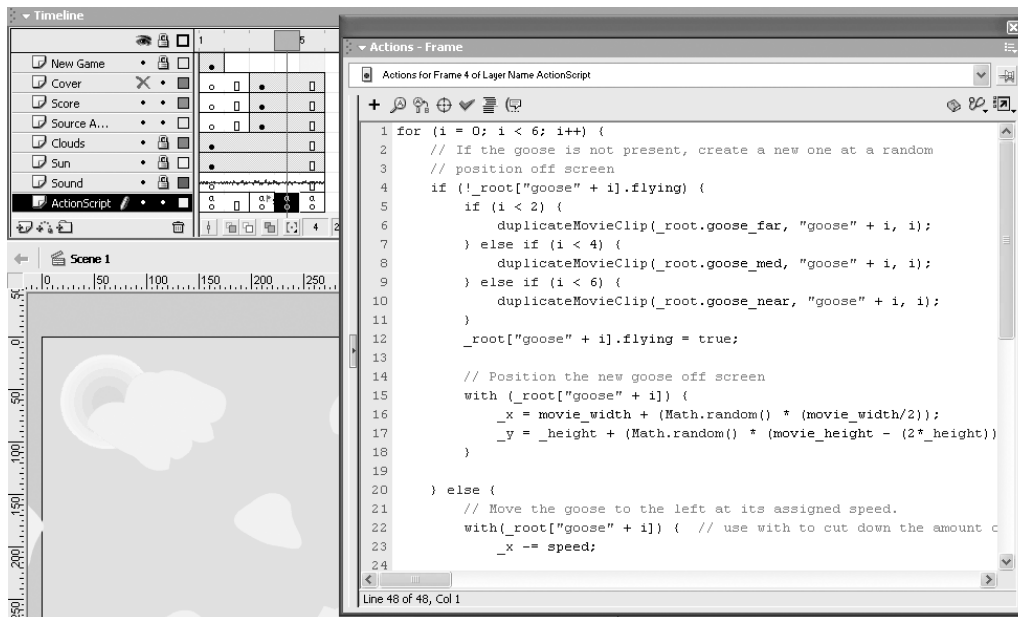


Figure 12.6 Adding goose creation and movement code

```

        duplicateMovieClip(_root.goose_far,
            "goose" + i, i);
    } else if (i < 4) {
        duplicateMovieClip(_root.goose_med,
            "goose" + i, i);
    } else if (i < 6) {
        duplicateMovieClip(_root.goose_near,
            "goose" + i, i);
    }
    _root["goose" + i].flying = true;

    // Position the new goose off screen
    with (_root["goose" + i]) {
        _x = movie_width + (Math.random() *
            (movie_width/2));

        _y = _height + (Math.random() *
            (movie_height - (2*_height)));
    }

} else {

```

```

        // Move the goose to the left at its set speed.
        with(_root["goose" + i]) {
            _x -= speed;

            // Remove the goose if it moves off the
            // left of the screen
            if (_x < -(_width/2)) {
                removeMovieClip( );
            }
        }
    }

    // Calculate the time, jumping to the
    // game over screen if they've finished.
    if (frames == fps) {
        time -= 1;
        timeleft = time;
        frames = 0;

        if (time == 0) {
            gotoAndStop("gameover");
        }
    } else {
        frames++;
        play( );
    }
}

```

This piece of code starts by using a “for” loop to loop six times, once for each goose that will appear in the game. It then checks to see if the goose exists by looking at the goose’s “flying” variable value. Once created and positioned, the “flying” variable is set to “true”, if it isn’t “true” then we need to create a new goose.

If the goose is not present, the appropriate goose is duplicated depending on its number. The first two geese are far away, the next two are medium distance and the final two are near. The duplicateMovieClip function is used to duplicate the movie clip and it is given the name goose#, where # is the number of that goose.

The “flying” variable for that goose is then set to “true” and the _x and _y positions updated to randomly position the goose off the right-hand side of the screen.

If the goose is present, i.e. it’s “flying” variable is set to “true”, then the goose is moved left by its specified amount. The amount is set as the goose’s “speed” variable. If the goose reaches the left-hand side of the screen it is deleted using the removeMovieClip function.

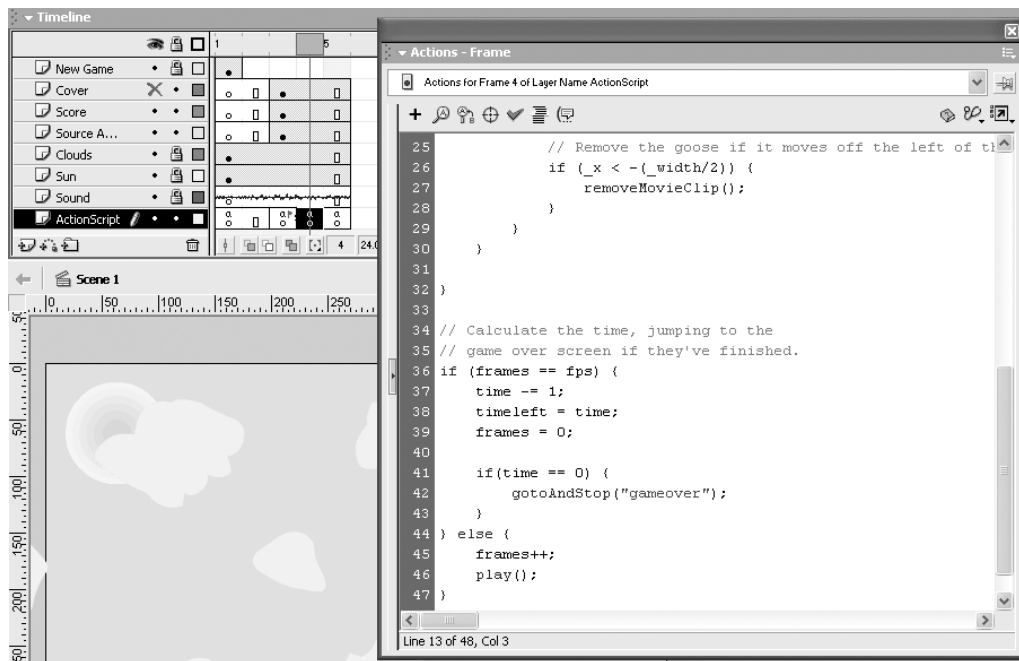


Figure 12.7 *Geese in flight*

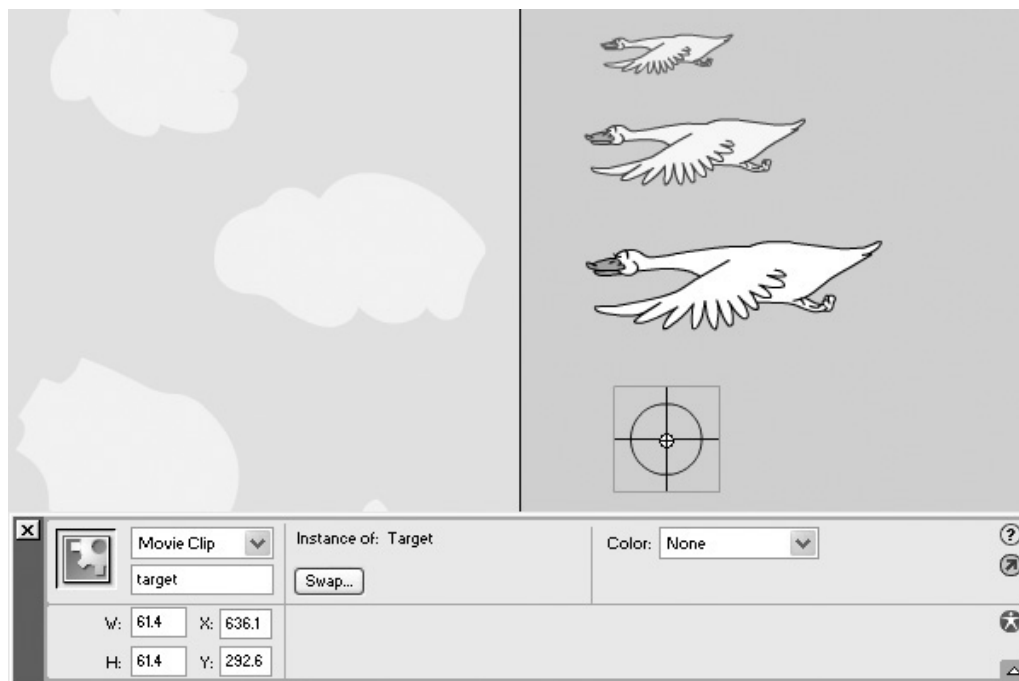


Figure 12.8 *Adding the target*

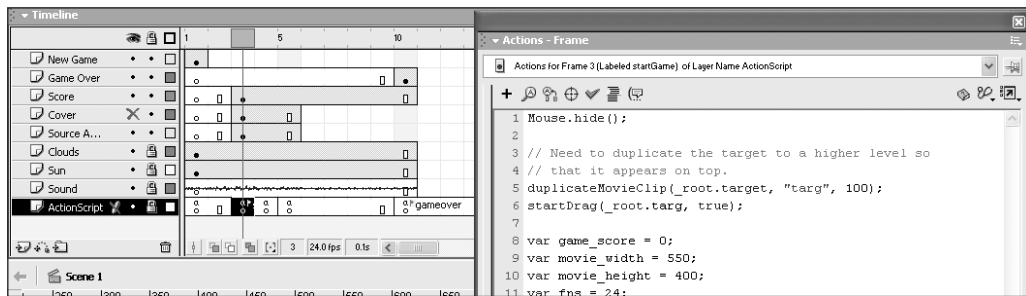


Figure 12.9 Code to move the target

The second section of this code is used to calculate and display how much time has elapsed. The calculation works by keeping a count of how many frames have passed. If the frames passed matches the movie speed (fps), then one second has elapsed. One is then deducted from the elapsed time and if the time reaches zero then its game over and the movie jumps to the “game-over” frame. Otherwise, the frame count is increased by one and the game continues.

Finally, create a keyframe on frame 5 of the “ActionScript” layer and add the following code repeat frame 4:

```
gotoAndPlay(4);
```

By compiling and running the movie you will see the geese flying across the screen.

Building the Game – Adding the Target and Finishing Touches

Now that we’ve got our geese we need to add a method of shooting them. To do this, add the “Target” movie clip from the library to the “Source Assets” layer on the third frame. Give the movie clip the instance name “target”; this will act as our target. To attach the target to the mouse movement, add the following code to the frame action of frame 2 on the “ActionScript” layer:

```
Mouse.hide();

// Need to duplicate the target to a higher level so
// that it appears on top.
duplicateMovieClip(_root.target, "targ", 100);
startDrag(_root.targ, true);
```

This code starts by hiding the mouse pointer, duplicating the target movie clip and attaching the target to the mouse using the startDrag function. Running the movie at this point you will see the pointer is replaced by a target.

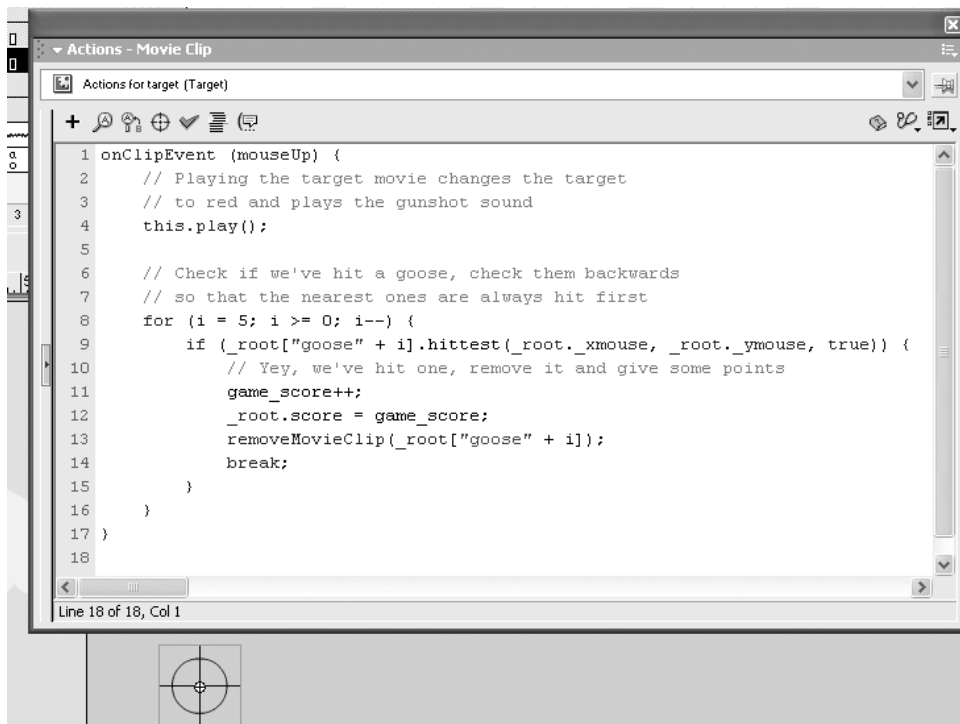


Figure 12.10 Adding target code

To add action to the target, use the following code on the movie clip action of the target:

```

onClipEvent (mouseUp) {
    // Playing the target movie changes the target
    // to red and plays the gunshot sound
    this.play( );

    // Check if we've hit a goose, check them backwards
    // so that the nearest ones are always hit first
    for (i = 5; i >= 0; i--) {
        if (_root["goose" + i].hitTest(_root._xmouse,
            _root._ymouse, true)) {
            // Yey, we've hit one, remove it and
            // give some points
            game_score++;
            _root.score = game_score;
            removeMovieClip(_root["goose" + i]);
            break;
        }
    }
}

```

This code uses the `onClipEvent(mouseUp)` to detect when the player clicks the mouse. It then plays the target movie clip to play a sound and checks through each goose on the screen to see if it has been hit. It uses the `hitTest` function to check if the goose has been hit; if it has, the player's score is increased and the goose movie clip removed.



Figure 12.11 Game over screen

All that now remains is to add an end of game screen. For this, create a keyframe at frame 10 on the “ActionScript” layer and label it “gameover”. On the frame action add the code:

```
Mouse.show( );
stopDrag( );
removeMovieClip('targ');

for (i = 0; i < 6; i++) {
    removeMovieClip('goose' + i);
}
stop( );
```

This code simply resets the game by showing the mouse pointer, removing the target and removing all the geese from the screen. Next, create a new layer named “Game Over” and create a new keyframe at frame 10. On it place the “Game Over” movie clip from the library. Beneath this place the “Play Again” button and on the action for this button add the code:

```
on (release) {
    gotoAndPlay('startGame');
}
```

Publishing and Exporting Your Macromedia Flash MX Movies

Once you have created your Flash movie it is now time to publish it. The Publish command creates all the files you need to deliver your movie on the web (including HTML files), or as a stand-alone file that can be played without having Flash Player installed, called a projector. You can also get Flash to publish a GIF Image, JPEG Image, PNG Image, Quicktime movie or RealPlayer SMIL file.

Optimization

Before you publish your movie for the first time, you should check to see how your users might actually view it on the web. When building your movies you have to bear in mind the issue of quality versus quantity, and Flash lets you find where your movie is slowing down or is not properly optimized by using the Bandwidth Profiler, and use the simulated streaming function to see how long a movie will take to load with different bandwidth settings.

To use the Bandwidth Profiler:

1. Open the gooseATTACK.fla file from your computer’s desktop and choose Control>Test Movie. The file will now launch in the Flash Player.
2. From the Debug menu choose 56K to simulate a 56K modem speed and then choose View>Bandwidth Profiler. The graph that now appears tracks the amount of data that is being transmitted against the timeline of the movie. The bottom line represents the amount of data that will safely download quickly enough to keep up with the frame rate of the movie.



Figure 12.12 *Bandwidth Profiler*

- Now choose View>Show Streaming. This will show you how quickly the movie loads, with a green bar at the top of the Bandwidth Profiler giving you a graphical display of the amount of progress made. The green bar stops as it passes over the higher bars as it takes longer to load these parts of the movie.

When you identify frames that are too large to stream efficiently, you may want to optimize your movie. You can optimize playback in a number of ways – here are a number of tips for optimization:

- Use symbols for every repeated element in your movie.
- Use alpha channels sparingly as they take a lot of processing.
- When you embed fonts select only the characters you need.
- Use tweened animation – this is less memory than frame-by-frame animation.
- Group elements wherever you can.
- Use movie clips instead of graphics for animated sequences.
- Use MP3 sound compression, and use sound sparingly.
- Limit the number of stroke types and the number of fonts.
- Use bitmaps sparingly.



Figure 12.13 *Publish Settings*

Publish Settings

Flash offers many publishing options, as mentioned previously. Publishing your movie is easy.

To publish your movie:

1. Open the your gooseATTACK.fla. Choose File>Publish Settings from the main menu.
2. In the dialog box that now appears select the Flash, HTML, GIF Image and Windows Projector check boxes. As you select the format a tab will appear at the top of the box.
3. Now click on the Flash tab – you will see a number of options, of which the following should be modified:
 - Load Order – Determines the order in which the layers in each frame will load. This is important when files are being downloaded from slow modems, as Flash shows each layer as it has been loaded. Top Down sets the frames to load from the top one first and vice versa. Set this to Bottom Up.
 - Generate Size Report – Make sure this is checked. It tells Flash to generate a text file that contains information on the size of each individual file which is very useful.
 - Protect from Import – Check this box. This allows you to protect against your SWF file being imported into Flash.
 - Version – Choose Flash 6 to enable the full functionality of your movie.

4. Click the Set button next to the Audio Event setting, a dialog box appears, allowing you to change the compression settings for the sounds in the movie, meaning any file that is not compressed will use these settings. Set the compression to MP3, the Bit Rate to 32 kbps and the Quality to Fast.
5. Now click on the HTML tab to set the HTML settings. First set the Template to Flash only. This setting determines what kind of template will be generated; you can get more information on the template by clicking the info button.
6. From the Dimensions menu, choose Percent to enable your movie to scale with the size of the web browser. Then disable the display menu option; this disables the shortcut menu that opens when the user right-clicks on the screen. Lastly set the Quality to High.
7. Now click on the GIF tab to see what options are available, but leave them at their default. Click Publish when you are ready to export your movie. Now look at all the movies you have just published. Open the Windows projector, which will work without Flash Player installed (the Macintosh projector works in the same way), so you can send it to a friend who has not got Player installed.

When is a Game Finished?

Finishing a game does not mean just getting to a point where the game is playable. A finished game will have an opening screen, a closing screen, menu options and information on how to play and start the game, introduction screens to playing, success screens and a scoreboard.

There is always a temptation to believe that once the bare bones of a game are complete then the game is finished. But this is not so and there is a process that you need to go through, which is made up of two stages. The first stage is the debugging of the game, this is done by getting people who have not been part of the project to test your game. The next stage is the finishing touches. The difference between when you thought the game was complete and when it is complete can be (depending on the size of the game) weeks if not months. As your game becomes more complex, you will get exponentially more involved in the finishing process. Accomplishment does wonders for self-esteem!

What you are looking for at the end of the game development cycle is a user's ability to pick your game up and have no problems moving through it, because everything is well presented. With your first game you will learn all about the details that go into really finishing a game, so do not miss out on the details of wrapping things up; it will come back to haunt you if you do not spend the time on it now.

I suggest you gradually build up your understanding of games development, and I have outlined the order of games development at the beginning of this chapter for you to learn about the process. This is not the only way, but it works, and it is so much easier to learn to walk before you start running.

As a developer, if you are really interested in making games, then you need to separate your desire to create the next cutting edge game, and focus on how your art is implemented within a gaming environment. After all, the design and creation of a simple game is hard, especially if you want to do it well. Once you have finished a game, then you're onto the first steps of creating games that utilize your skill as a developer. Just remember one thing – if you can't play it, then it's not a game. I would love to see how you customize this game, so please send me an email (alex@sprite.net) with a link to what you have done.

Start Very Small and Work Your Way Up

Until you understand that all the skills in game development are learned by experience, you will probably be doomed never to finish your projects. The learning is incremental. Unless you start small and build up you will make many attempts before you manage to finish a full-scale epic, and the problems will be disproportionate to its size.

13 ActionScript-driven Effects and Techniques

This chapter deals with ActionScript-driven effects, and the techniques that go into building code. The chapter consists of 18 examples, all of which can be plugged into your movies to create useful effects, but at the same time taking a minimum of file size. Use this chapter to learn how to add a whole new range of effects and interaction to your movies and explore some of the more complex aspects of ActionScript. All the code and FLAs are available for download from www.sprite.net/understanding.

The Flash timeline allows you to create very sophisticated effects without having to deal with the code that lies behind them. However, sometimes there are benefits to getting in there and using code to generate your effects. ActionScript-driven effects are fundamental to Flash games, and can generate some amazing text effects and dynamic animations that the user can control! The developer can also benefit from ActionScript, using code to automate repetitive tasks.

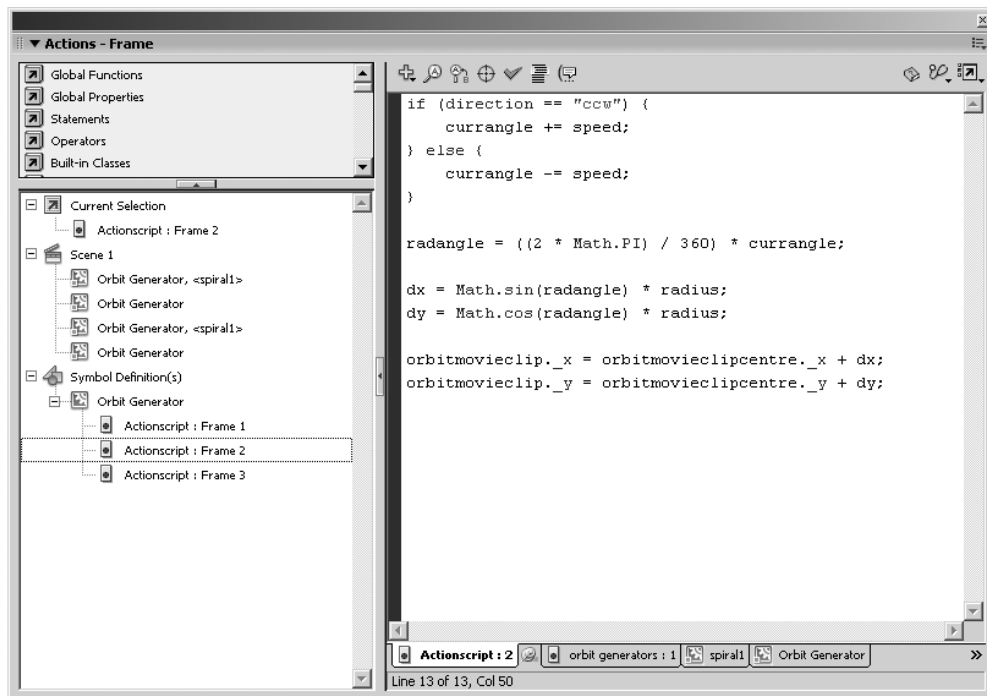


Figure 13.1 Code driven animation

The development of code-driven animation, particularly in Flash, has opened up whole new realms of possibility for designers and programmers alike, and we have recently seen a move away from the conventional tweening techniques towards code-based techniques. Code-based techniques can produce a wide range of results, ranging from the artistic to the mechanical; the results are always computer generated but with a little imagination can be stunning. Traditional frame-by-frame animation can also be combined with code-driven animation to make your animations run more smoothly and create unique effects, and you can also easily model physical behaviors, such as gravity. Using code-driven animation, animators are now not limited to traditional animation techniques, and can create complicated effects with just a basic knowledge of the methods used. Code-driven animation can be as small as 3% when compared to a similar frame-driven animation. This is great for delivery over a telephone line or playback on a PDA.

Before Flash, code-driven animations had to be written in lower-level languages such as C and Java, which needed a lot more programming expertise, were less robust (particularly Java) and took a lot longer to write. With the development of ActionScript, a high-level language, a lot of the hard work is already done for you – for example, you do not have to tell it how to move an asset, you just have to tell it where to put it. ActionScript also has a robust structure to code around (the Flash Interface) which makes it much easier to get to grips with.

The effects consist of a number of static elements that use a function to move over a mathematically generated path. The beauty of code-driven effects lies in its flexibility, and the parameters of the animation can easily be changed to manipulate your animation without having to redraw it. Code-driven animation is also very memory-friendly, and only requires the space needed to store the lines of code. You can also reuse lines of code in other code-driven animations, making them very efficient. The fact that you can reapply your code also means that you can easily use a favorite technique over and over without having to redraw all the frames. Our examples are a source of extra functionality for your animations, and give sophisticated results with minimum fuss and effort.

You can experiment with the examples in the rest of the chapter. Remember you can edit these in any way you like to fit them into your movies. Have fun!

The ActionScript-driven Effects Movies

- **Drag and Drop on Grid** – dropping an object so it snaps to a grid.
- **Moving Eyes** – eyes that follow your mouse pointer around the screen.
- **Cursor** – this shows you how to change the appearance of your cursor, but with a difference!
- **Depth Swapping** – a draggable object that swaps depth as it moves along.
- **Wobble Buttons** – make your buttons wobbly.
- **Parallax Scrolling** – we demonstrate how to create parallax effects.
- **Magnifying Glass** – a great magnifying glass example to magnify the background of your movies.
- **Disney Sparkle** – the classic Disney wand sparkle, attached to your cursor.
- **Sparkler Effect** – creates fireworks that follow your mouse pointer around the screen.

- **Starfield** – a classic space starfield effect that you can plug into your movies.
- **Bubbles** – plug this effect into your animation to create bubbles rising to the top of the screen.
- **Bug** – create bugs that crawl around the screen.
- **Dancing Lights** – a nice animated effect to generate fading lights all around your screen.
- **Earthquake** – make the screen shake like an earthquake.
- **Gravity Footballs** – real gravity created by ActionScript.
- **Movie Controller** – add a dynamic controller to your movies.
- **Orbit** – a simple way to create complex orbit effects.
- **Snow** – a plug-in and play snow effect.

Drag and Drop on Grid

This is the full effect of the `startDrag()` method with a snapping functional that snaps to a grid. The movie clip tile has this code attached to it:

```
on(press){
    startDrag(this, false, 0,0,_root.screenWidth-_root.mcWidth,
    _root.screenHeight-_root.mcHeight);
}
```

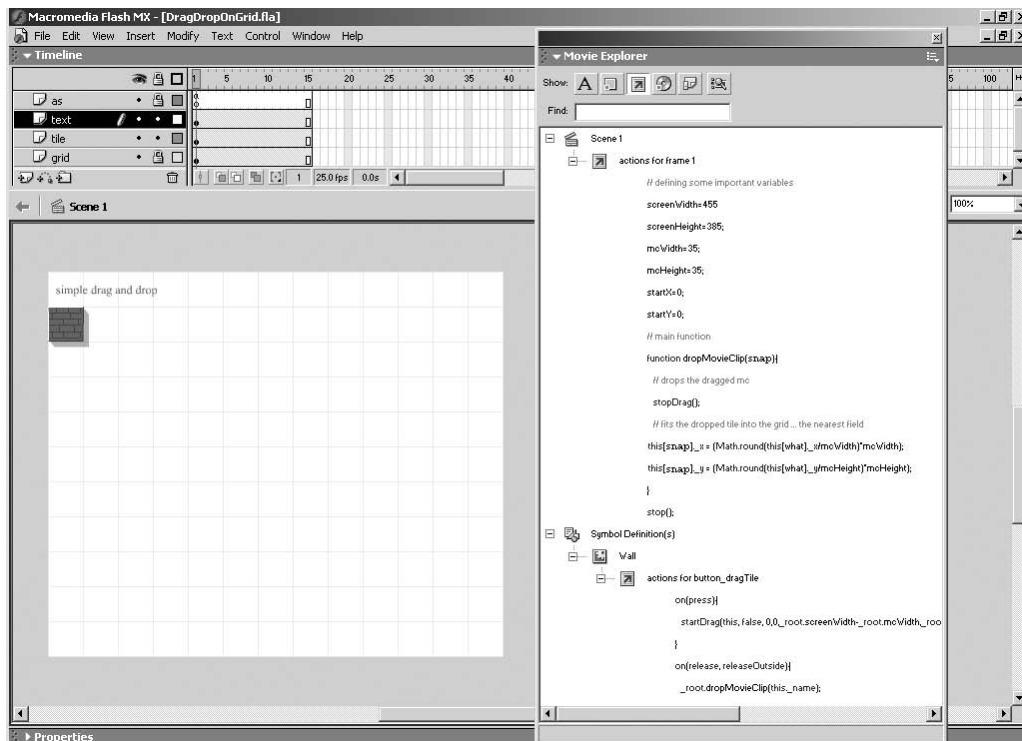


Figure 13.2 You can download the *DragDropOnGrid.fla*

```
on(release, releaseOutside){
    _root.dropMovieClip(this._name);
}
```

What is important to remember here is that the tile movie clip is divisible exactly into the total size of the stage.

Moving Eyes

A fun pair of moving eyes that follow your mouse pointer around the screen. See *MovingEyes.fla* from the Sprite website, www.sprite.net/understanding. These eyes are part of a whole face, which can be edited in any way you like simply by editing the “face” graphic. The eyes are a movie clip called “creepy eyes” that can be dropped into your movie to make them work. The ActionScript that drives the example is quite complicated, and can be found on the actions layer of the “creepy eyes” movie clip. It uses trigonometry to track the cursor and point the pupil of the eye towards the pointer, whilst at the same time keeping it inside the boundary of the eye.

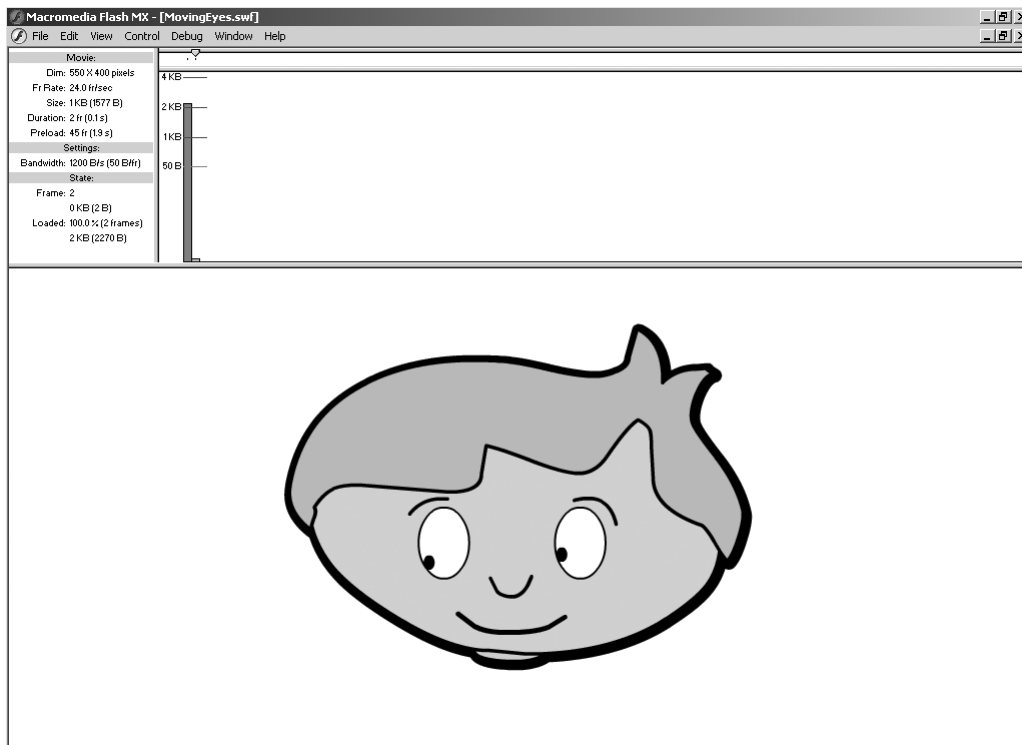


Figure 13.3 You can download the *MovingEyes.tif*

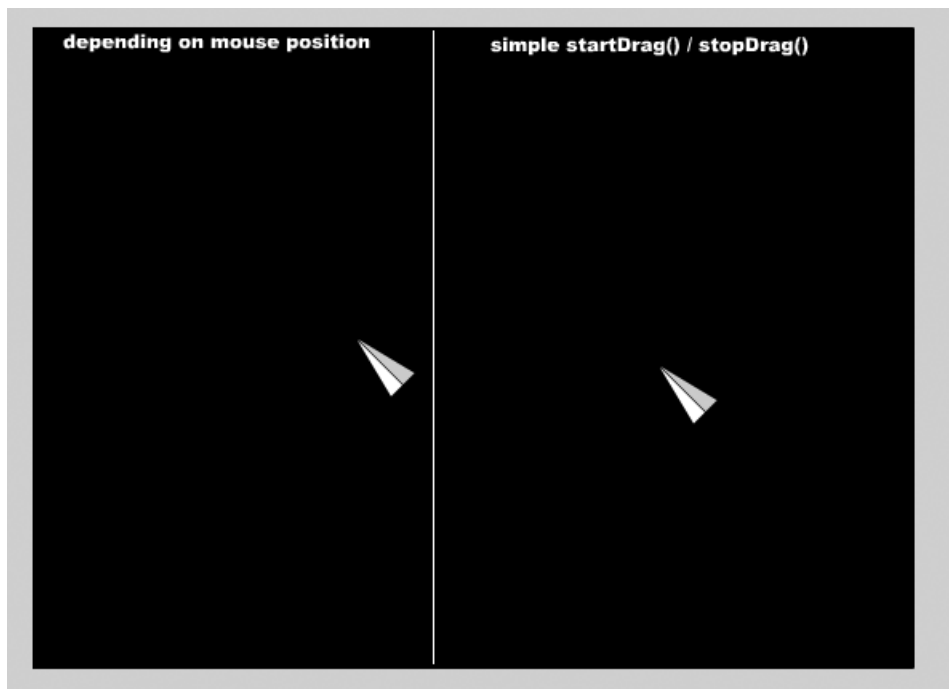


Figure 13.4a The stage is split into two to demonstrate the effect

The movie clip is very customizable but your goal should be to achieve the level of understanding of ActionScript that allows you to rewrite bits of this code for different applications.

Cursor – Change of Cursor

Changing the appearance of the cursor is a great way to make your movies and animations more visually appealing. It is also very simple. All you need to do is drop the “Change Cursor” movie clip into your movie, and the cursor will change. Updating the graphic in the “Change Cursor” movie clip. But what I have done here is demonstrate two ways of doing this. When the cursor is in one half of the stage it does the conventional `hide()`, and when it is in the other half of the stage it uses

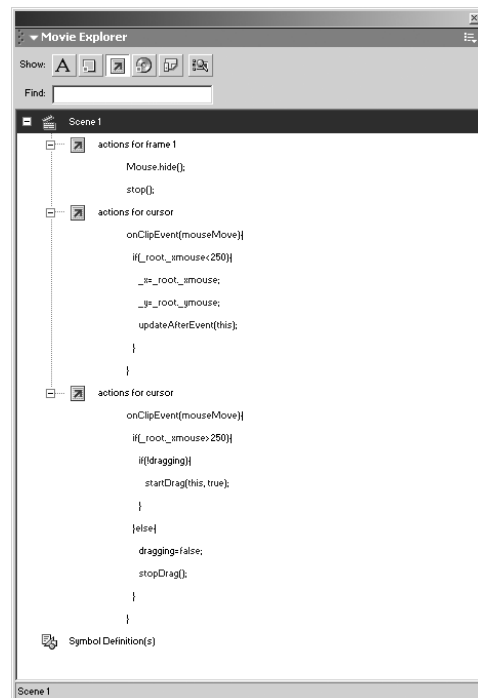


Figure 13.4b All the code to run both tests

the `updateAfterEvent()` method. The ActionScript in this example, like all the other examples from this book, is available from the Sprite Interactive website www.sprite.net/understanding. The `Cursor.fla`, first of all, hides the mouse, and then locks the “cur” movie clip onto the mouse, it then uses the `startDrag` command to drag that movie clip around the screen. The code looks like this:

```
Mouse.hide( );
cur._x = _xmouse;
cur._y = _ymouse;
startDrag( cur, true );
```

When you move to the other half of the stage you use the `updateAfterEvent()` method, which gives you a smoother transition without the shadowing effect in the `startDrag()` method.

```
onClipEvent(mouseMove){
    if(_root._xmouse<250){
        _x=_root._xmouse;
        _y=_root._ymouse;
        updateAfterEvent(this);
    }
}
```

Depth Swapping on Drag of a Movie Clip

As you pick up the `dragMe` movie clip and move it across the y -axis the movie clip is scaled by dividing the distance across the y -axis by 10 and then multiplying it by 4. This is a very simple effect that can be used in both interface design as well as narrative. The thing that has the biggest visual effect is the depth swapping, which seems to send things from the foreground to the background. Any object in the middle of the screen will seem to be sent to the background as it moves down in its y value. This happens because `this.swapDepths(this,_y)` maps directly onto the value of its depth.

```
onClipEvent(enterFrame){
    // swapping the depth of this movieclip
    this.swapDepths(this._y);
    // scale this movieclip depending to the y coordinate
    this._xscale=this._yscale=(this._y/10)*4;
}
```

Wobble Button

The wobble button is a function that creates wobbly buttons. Buttons are essential if you are building interactive movies, where the user can either navigate around or make decisions on screen. The wobble button is a fun tool that would be perfect for creating fun buttons on a website or on a kids' game. You will see the effect of the wobble button when you run your mouse pointer over it. Have a look at `WobbleButton.fla` from the website.

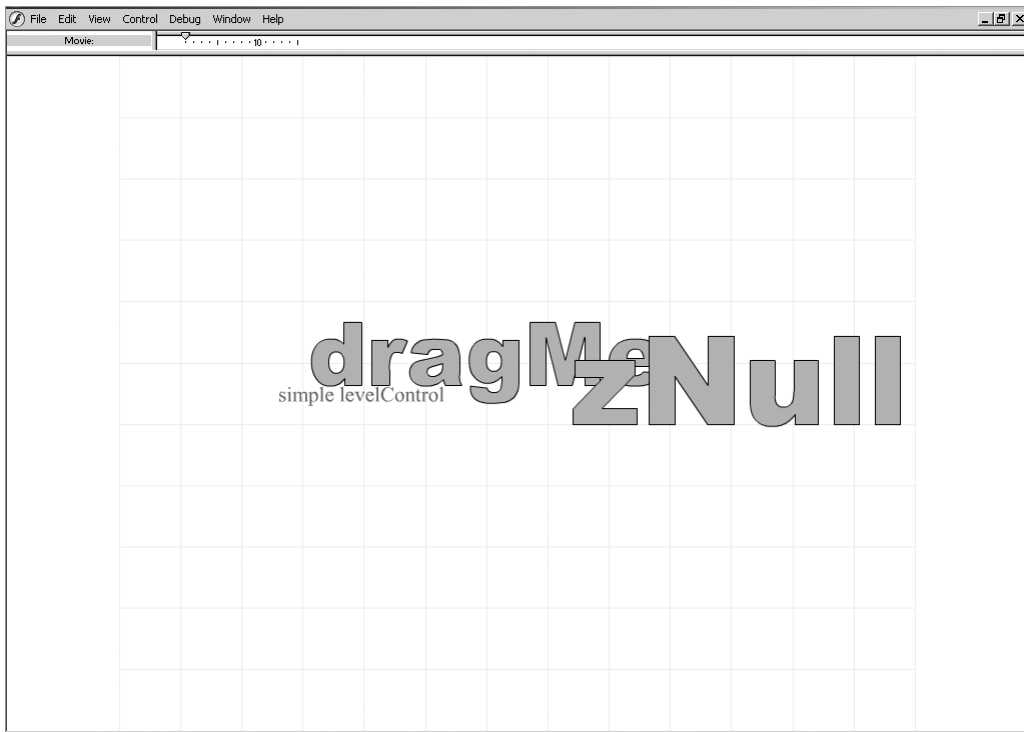


Figure 13.5 *Scaling movie clip as it moves across the y-axis*

You can also easily change the graphics that make up the button by just going into the actual button and changing the graphical assets. The script that runs the wobble buttons is fairly simple; it sets a number when you roll over the button that it increases its size to, and then sets another number as you move the mouse pointer away from the button to decrease the size of the button. The numbers to set the size of the button as the mouse rolls over it can be found in the “Wobbler” movie clip in the actions for the button, and you can experiment with changing them here. The actions on the “Wobbler” button are also where you put the interactivity for the button, and you can see the comments in the script explaining how to add interactivity to the on(release) handler. The most important thing to remember is that you have to reference the `_root` of the movie if you are adding interactivity, as the Wobbler button is a button within a movie clip within another movie clip and therefore is not on the top level of the main movie. The script is shown in Figure 13.6.

Parallax Scrolling

Parallax scrolling is the effect one sees when looking at a scrolling series of “layers” scrolling at different speeds. Each layer represents a distance from the viewer (the foreground, the background, far in the distance, etc.). As the entire view “scrolls” or moves in a two-dimensional direction (no depth movement), the layers scroll at a different speed. The farther away from the viewer a layer is supposed to be, the slower it will scroll. This creates the illusion of depth. Parallax

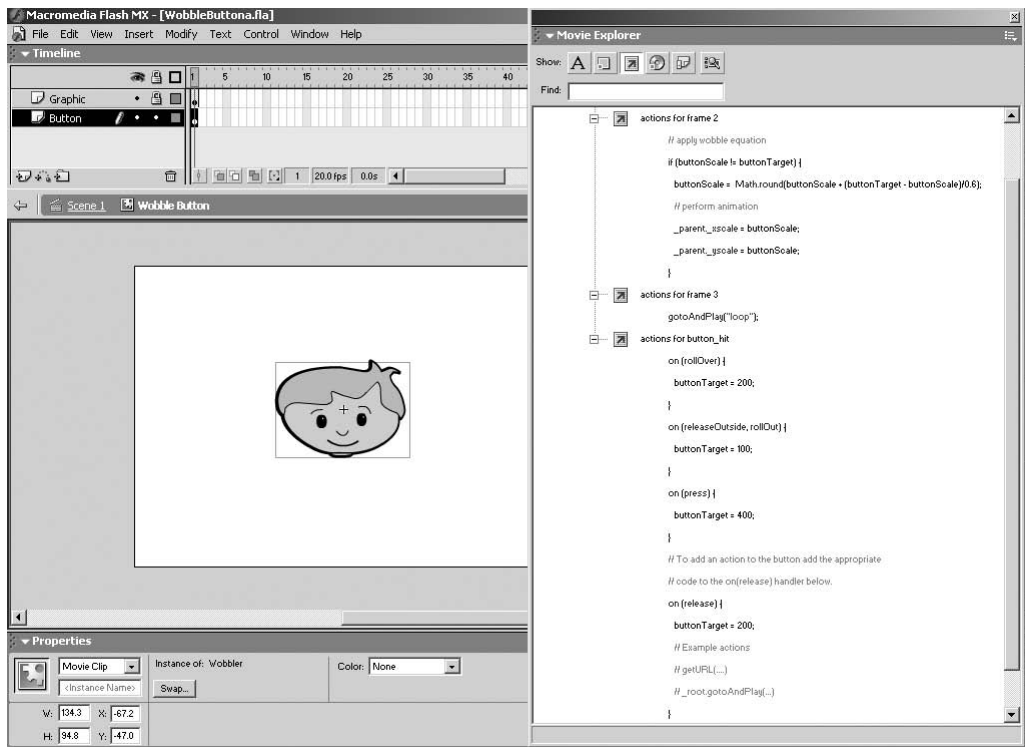


Figure 13.6 Most of the actions are attached to the button

scrolling is used extensively in computer games, you can see the example in Parallax.fla from the Sprite Interactive website.

The beauty of parallax is that you can place a movie clip of a character moving in the foreground, and then change how fast it moves simply by speeding up or slowing down the speed of the scrolling. The example in this chapter uses a desert scene that can easily be modified; all you need to do is change the graphics in each movie clip that makes up the scene. There are four movie clips, Background, Far, Middle and Near, and you do not have to change the code to change the

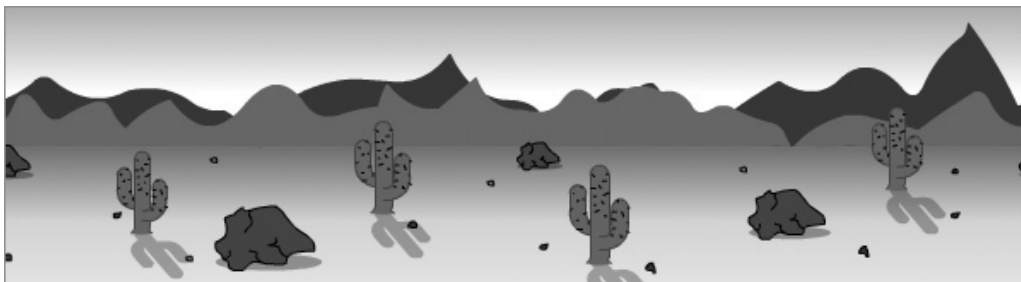


Figure 13.7 The Background, Far, Middle and Near move at different speeds

background, just the graphics. You will notice that the three levels of the scene all move at different speeds; this variable is set in the first frame of the Parallax Background movie clip. The movie clips are tiled across the screen, and the second frame of the movie clip sets each level of the scene to start its movement again if it has moved halfway across the main scene. The script to drive the scrolling is below.

```
// If the far background has moved half way, start it again
if (bg._x < -(bg._width/2)) {
    bg._x = -far_speed;
} else {
    bg._x -= far_speed;
}

// If the near background has moved half way, start it again
if (bg1._x < -(bg1._width/2)) {
    bg1._x = -mid_speed;
} else {
    bg1._x -= mid_speed;
}

// If the foreground has moved half way, start it again
if (bg2._x < -(bg2._width/2)) {
    bg2._x = -near_speed;
} else {
    bg2._x -= near_speed;
}
```

Magnifying Glass

A magnifying glass is a fun tool that you can feature in your animations. For example, it could be used for a game where the user has to find certain hidden objects around a picture using the magnifier. In the example we have provided you can magnify the Ellen and Zak website homepage.

The magnifier works by taking the picture you are magnifying and putting it through the lens, which acts as a mask over the first image, blowing the image up to twice its size and creating a magnified effect. It is quite a simple process, but is hard to get the picture positioning exact, so you can't move the picture around. To update the picture being magnified, all you need to do is go into the library, select the image, which in this case is image.jpg, and bring up its Properties dialog box, then select the Import button and choose the image you would like to magnify – the size of the image should be 640 × 480. Check out Magnify.fla on the website at www.sprite.net/understanding.

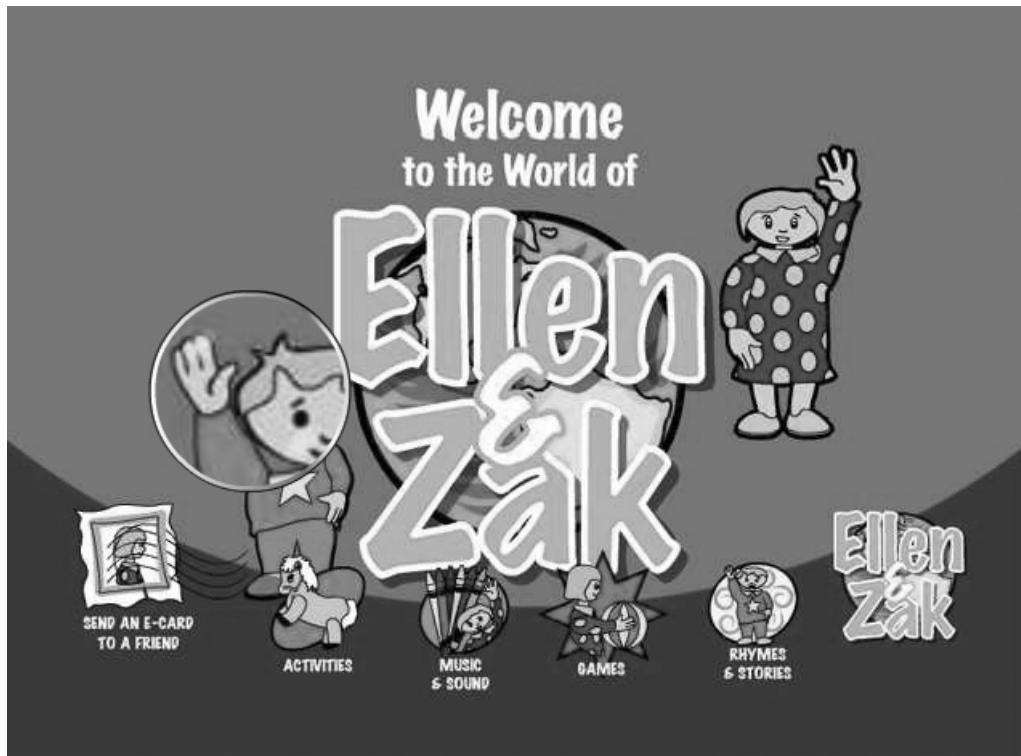


Figure 13.8 *Picture with scaled-up view through lens*

The ActionScript featured in the movie is very simple, and consists of a small piece of script to tell the movie to make the cursor invisible and to make the lens draggable, and actions in the lens movie clip to set two variables that tell it to magnify the picture by a factor of 2:

```
magimage._x = -(_parent.lens._x * 2);
magimage._y = -(_parent.lens._y * 2);
```

Sparkle

This animation creates a sparkler effect around the mouse pointer that follows it around and slowly dissipates towards the bottom of the screen. The sparkles generated are a random size and fall away from the pointer at a random speed, which are both fully customizable. To change the sparkle graphic open the “Sparkle Graphic” movie clip, and make any graphical changes you want. The sparkles appear at a set rate, which is not updatable, and they fall at a set speed. To integrate the sparkler with your movie all you need to do is drag the “Sparkler” movie clip from the library onto the stage and it will automatically work. The variables that dictate what the Sparkler looks like are shown in Figure 13.9, and can be found in the first keyframe of the “Sparkler.fla” movie clip on the website.

Disney Sparkle

This little piece of code-driven animation creates fireworks that follow the mouse pointer around the screen; check out Sparkle.fla on the website (www.sprite.net/understanding). All you need to do to integrate the effect into your movie is to drag the Fireworks movie clip onto the stage and it will automatically work. The fireworks are created in a movie clip called “animated firework” that can be found in the Firework Assets folder in the library. The “animated firework” clip can be modified so that the mouse pointer “sparkles” in any way you like, all you need to do is open the Fireworks movie clip in editing mode and change the graphic asset that it tweened in the small animation loop.

The ActionScript in the movie duplicates the Firework movie clip and sets it to play at a random angle, therefore creating the “sparkling” type effect. This ActionScript can be edited, but it is so simple and so effective that it is not really necessary. It is as follows:

```
startDrag ("drag", true);
fireworks = 20;
spark_num = "1";

spark._x = drag._x;
spark._y = drag._y;
duplicateMovieClip ("spark", "spark" + spark_num,
spark_num++);

if (spark_num < fireworks) { gotoAndPlay (2); }
```

Starfield

A starfield is a nice effect that can add a great sense of depth to your movies if you are building space animations. Check out Starfield.fla on the website (www.sprite.net/understanding); the “Star Field” movie clip can be dropped straight onto your movies, and only requires a small amount of

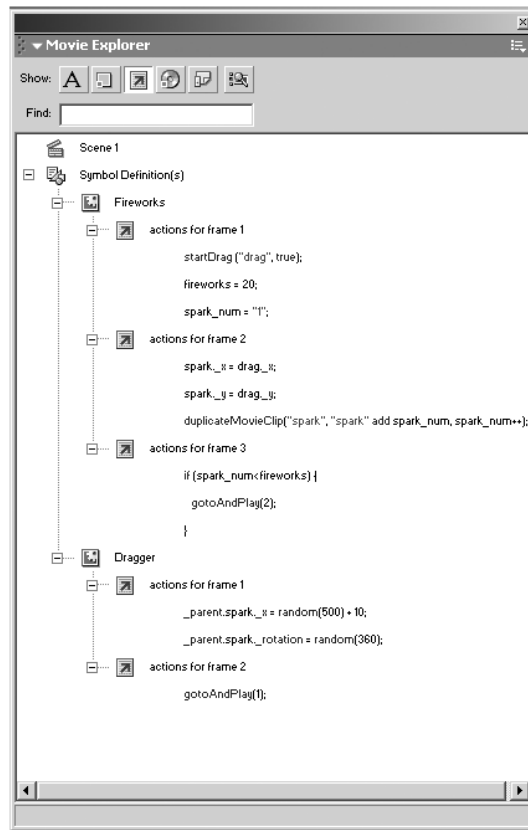


Figure 13.9 Picture with all the code for this movie in the Actions panel

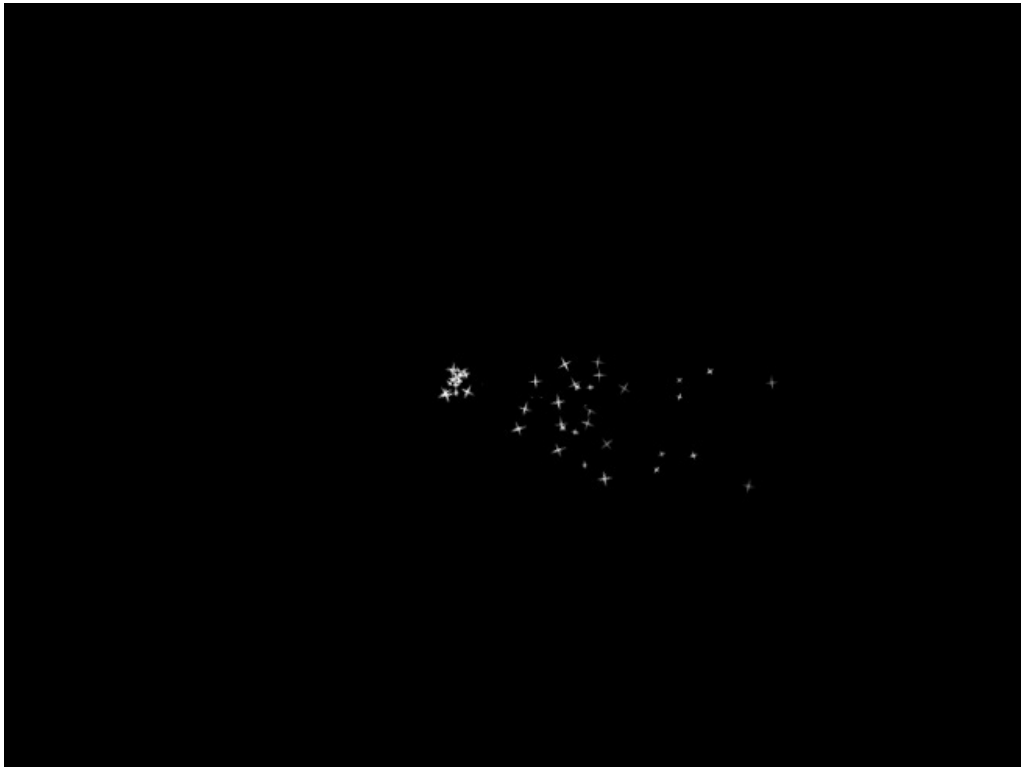


Figure 13.10 The “*animated firework*” clip

tweaking to work in any size of movie. The star graphic is a tweened animation that is called “Moving Star”, which can be found in the “Star Field Assets” folder in the library. You can change the graphical assets in this animation to make the stars in the starfield appear however you would like.

To see the ActionScript that drives the movie, open the “Star Field” movie clip in editing mode and then select keyframe 1 of the actions layer. Here you need to set the width and height of the movie you will be dropping the starfield into, and the rest of the calculations are done for you. The ActionScript looks like this:

```
// Set the width and height of your
movie here
moviewidth = 550;
movieheight = 400;

//-----
n=0;
```

```

// calculate global center point
centerPoint = new Object( );
centerPoint.x = Math.round(moviewidth/2);
centerPoint.y = Math.round(movieheight/2);
globalToLocal(centerPoint);

if (n == 50) {
    n = 0;
    removeMovieClip('star' add n);
}

duplicateMovieClip('star', 'star' add n, n);
this['star' add n]._x = centerPoint.x;
this['star' add n]._y = centerPoint.y;
this['star' add n]._rotation = random(360);
n++;

star._visible = false;

```

Bubbles

The bubble effect is very similar to the snow effect, but creates bubbles that rise up the screen and increase in size and fade out as they approach the top of the stage. Like the snow effect, you can drag and drop the Bubble Generator from the library onto the stage to integrate the effect into your movie. Download Bubbles.fla from the website.

The ActionScript that drives the bubble effect works in the same way as the snow effect, and you can change the variables in the same way. The ActionScript to control the bubbles can be found in the first keyframe of the actions layer of the Bubble Generator movie clip. The `number_of_bubbles` variable sets the number of bubbles that appear on the screen at once; the `bubble_speed` controls how fast the bubbles move up the screen, and you need to set the dimensions of the movie you are inserting the bubbles into to make the effect work properly. You can also change the sine values to affect how the bubbles travel up the screen. The ActionScript that you can edit looks like this:

```

number_of_bubbles = 50; // This sets the number of bubbles
bubble_speed = -2; // This sets the speed that the bubbles rise

movie_width = 550; // Movie dimensions
movie_height = 400; // Movie dimensions

// Precalc SIN values, speeds things up a little
sinTable = new Array(movie_height);

```

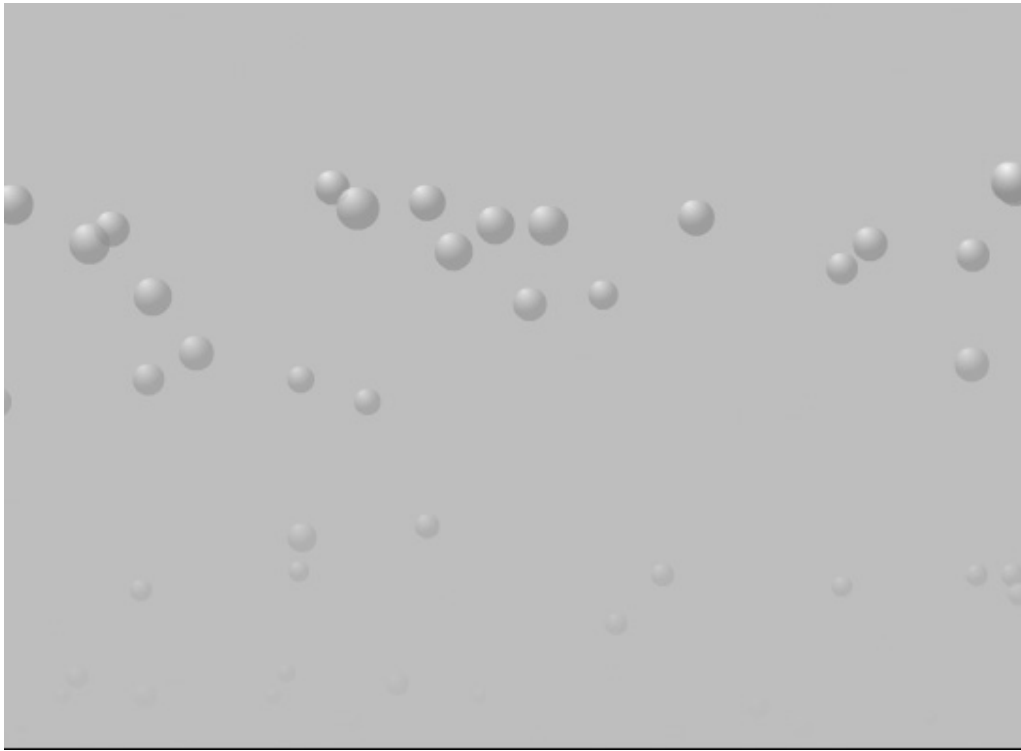


Figure 13.11 *Bubbles rise up the screen and increase in size and fade out*

```
for (i = -movie_height; i < movie_height; i++) {  
    sinTable[i] = Math.sin(i/16);  
}
```

Animated Bug

This fun code-driven animation allows you to put bugs into your movies that move around the screen in a random manner; it is called *Bug.fla* on the website. The bug graphic is a simple two-frame animation of the bug moving, and the ActionScript in the “crazy bug” movie clip makes the bug move around screen in a random way. It is easy to update the bug graphic – just open the “bug” movie clip in editing mode and edit.

The ActionScript that powers the bug can be found in the “crazy bug” movie clip, in the “actions” layer. The first keyframe controls the random movement of the bug, and you can set the speed, which controls how fast the bug moves, the craziness, which controls how crazy the bug’s movement is (how often it changes direction) and the movie width and height, which controls the bounding box for the bug’s movement. The ActionScript looks like this:

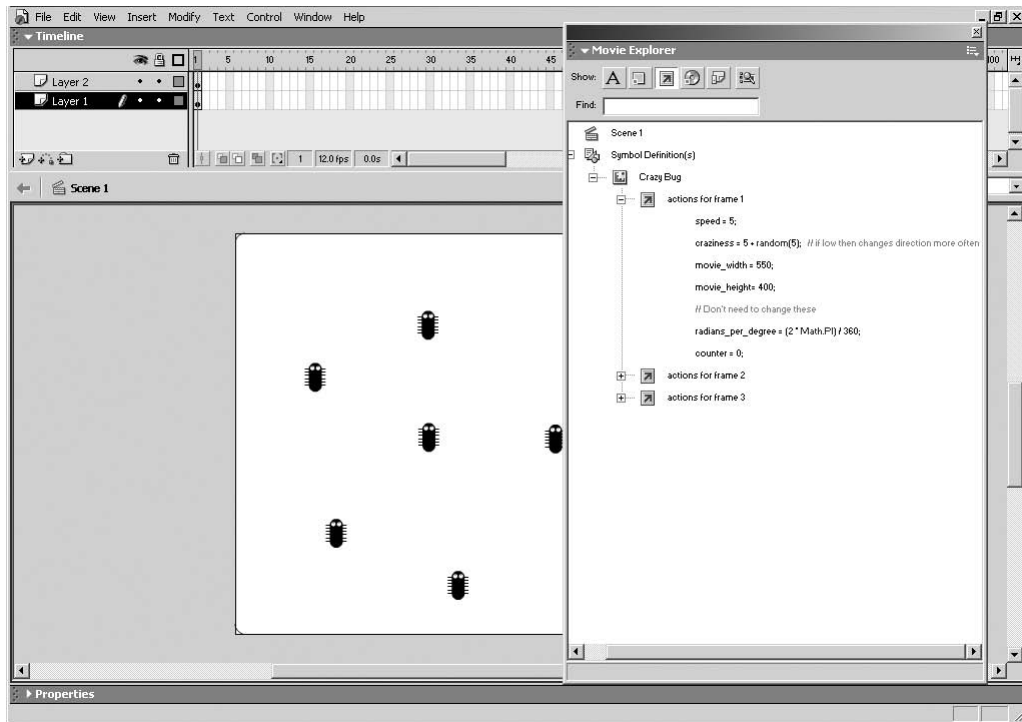


Figure 13.12 The “crazy bug” movie clip

```
speed = 5;
craziness = 5 + random(5); // if low then changes direction more often
```

```
movie_width = 550;
movie_height = 400;
```

```
// Don't need to change these
radians_per_degree = (2 * Math.PI) / 360;
```

```
counter = 0;
```

Dancing Lights

This is a great piece of code-driven animation that generates a preset number of lights that dance and fade into the distance; this could be used as an introduction for an animation, or to add a visually appealing effect to your animation. Look at *DancingLights.fla* on the Sprite Interactive website www.sprite.net/understanding. To use the dancing lights as an effect all you need to do is drag as many instances of the “Dancing Lights” movie clip onto the stage as you want, they will then play when the playhead passes over the movie clip.

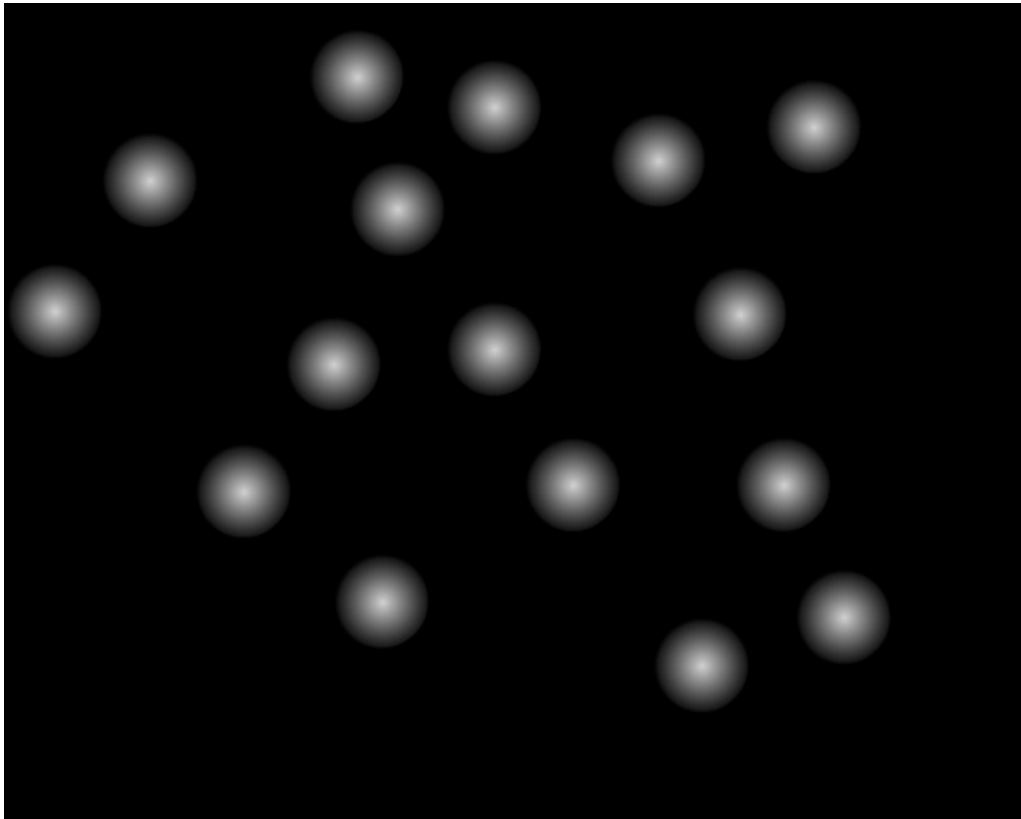


Figure 13.13 *A preset number of lights that dance and fade*

The graphics of the dancing lights can easily be changed by opening the movie clip in editing mode and substituting the “dust” graphic for your own. The ActionScript that drives the lights can also be edited. You will find this script in the Actions layer of the “Dancing Lights” movie clip – the first keyframe sets the direction of the light, the x and y values, and also sets the lifespan of the lights; the higher number you enter here, the longer it takes to “die”. The rest of the script in the other two keyframes of the actions layer generates random behaviors for the lights, so there is no need to edit these as the results are random anyway. The actions you can edit look like this:

```
vx = (random(3)-1)/5; // Give the light a random direction
vy = (random(3)-1)/5; // Give the light a random direction
life = 70; // Set number of loops particle will live for
```

Earthquake

This is a handy piece of ActionScript that makes your movies “quake” when it is placed on the stage with them, great if you want to simulate an earthquake effect. Have a look at Earthquake.fla on the website. To use this, all you need to do is place the “quake?” movie clip into your

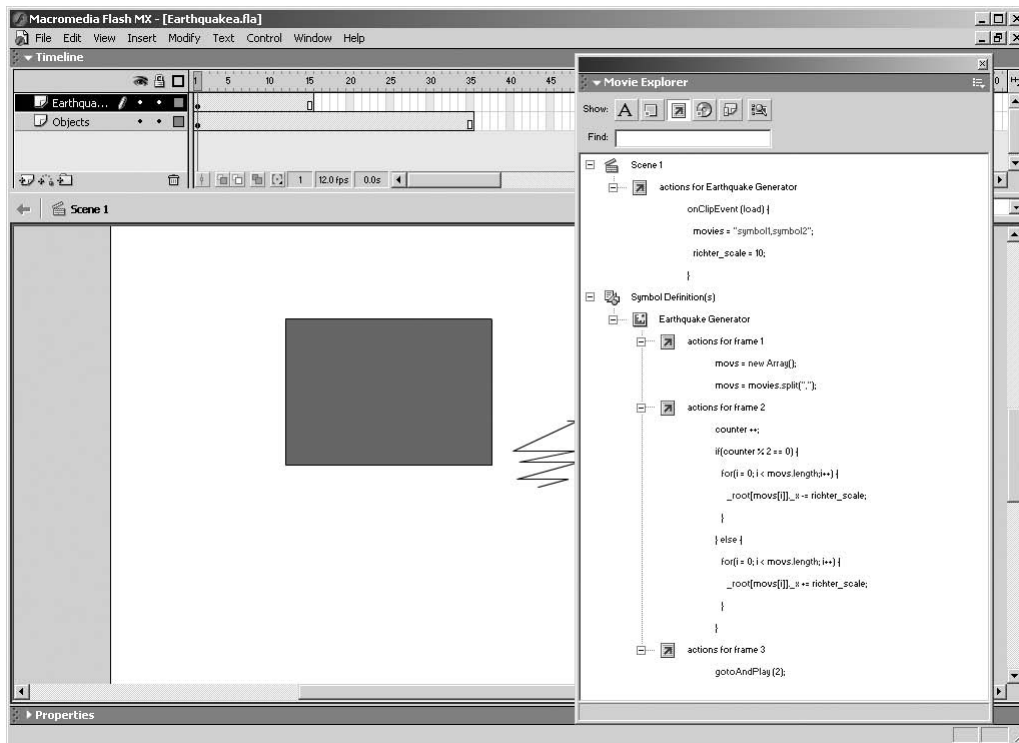


Figure 13.14 The “Earthquake Generator” movie clip

animation and then it will work automatically. The ActionScript that you can edit can be found on the “Earthquake Generator” movie clip, which looks like a zig-zag. On this clip you have to tell the generator which movie clip to “quake”; this has to be the instance name of the clip, which you set in the Properties panel, and you also have to specify the `richter_scale` of the earthquake – the higher this setting is, the faster the object will vibrate.

Gravity Footballs

Creating a gravity effect using ActionScript is a great way to animate objects without using complicated motion guides or paths. You can add this football to your movie simply by dragging the “Football” movie clip from the library onto the stage, and you can easily change the graphical appearance of the football object by changing the graphic asset in the “Ball” movie clip that is situated in the “Football Assets” folder.

The ActionScript that powers the animation is found in the first three keyframes of the actions layer of the “Football” movie clip. If you open the actions for the first keyframe of the Actions layer you will see all the editable areas. You can set the vertical velocity of the ball, to control how fast it falls towards the ground, the horizontal velocity of the ball, which runs from left to right, so

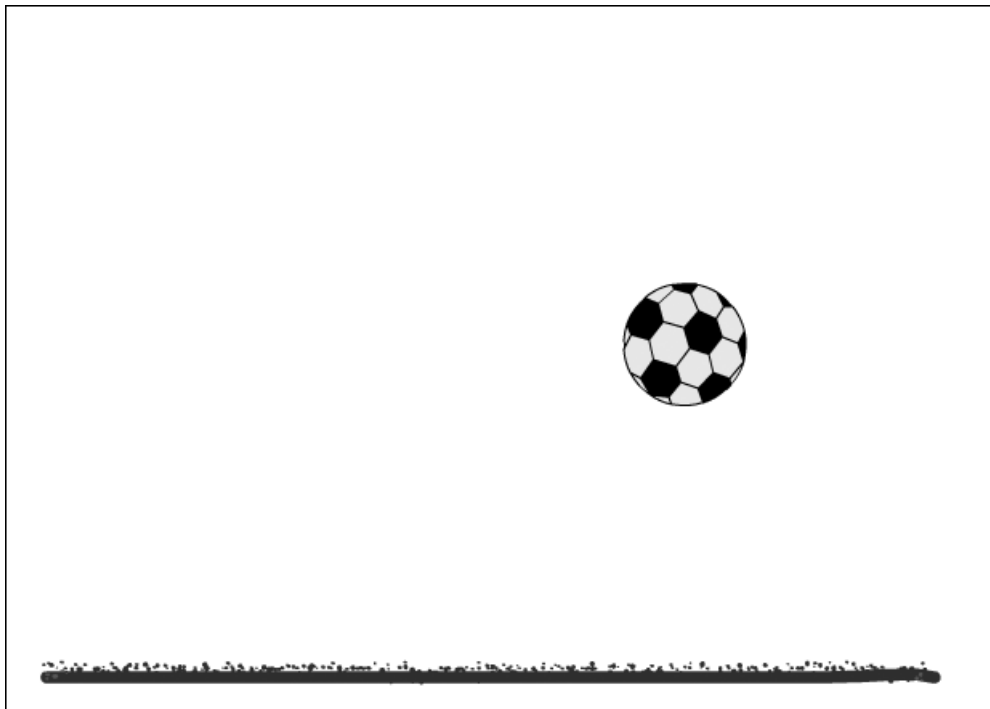


Figure 13.15 Have a look at *Gravity.fla* on www.sprite.net/understanding

if you want the ball to travel right you have to enter a minus figure (as you can see from the default entry). The next area sets the radius of the ball, and then the strength of the gravity; try entering different settings here to see how gravity affects the ball. The bounciness of the ball is controlled in the next part of the script, ranging from 0.1 to 1.0, and the last area of editable script is to set the movie dimensions, which make sure the football fits in with the movie you are dropping it into. The script looks like this:

```
// Vertical velocity
yv = 0;

// Horizontal velocity
xv = -2;

// Radius of ball
radius = ball._height / 2;

// Gravity
gravity = 10;

// How bouncy the ball is 0 - 1          (really bouncy)
bounciness = 0.8;
```

```
// Movie dimensions
moviewidth = 550;
movieheight = 400;
```

Movie Controller

The movie controller is a simple movie clip that can be added to any movie, and allows you to dynamically control the movie while it is playing. A movie controller is particularly useful if you would like the user to be able to control an animation while it is playing. It could be used, for example, if you have written a presentation in Flash that you would like the recipient to be able to control. You can use the movie controller in any movie simply by dragging it from the library and positioning it on the stage. The movie controller uses all the scenes and frames in the movie to calculate the position of the playhead, and if you want to use it in a movie with multiple scenes you will have to place it in every scene of the movie. Check out `MovieController.fla` on the website.

The movie controller works first by calculating the number of frames there are in the movie, and applying these to the length of the dragbar to the left of the controller. The controller then sets up the actions for each of the buttons on it – backwards, rewind, fast forward and pause – and then tells the dragbar how to work out which frame to go to when it is dragged (the current frame of the movie multiplied by the `pixels_per_frame` variable). To tell each button to work, each one has

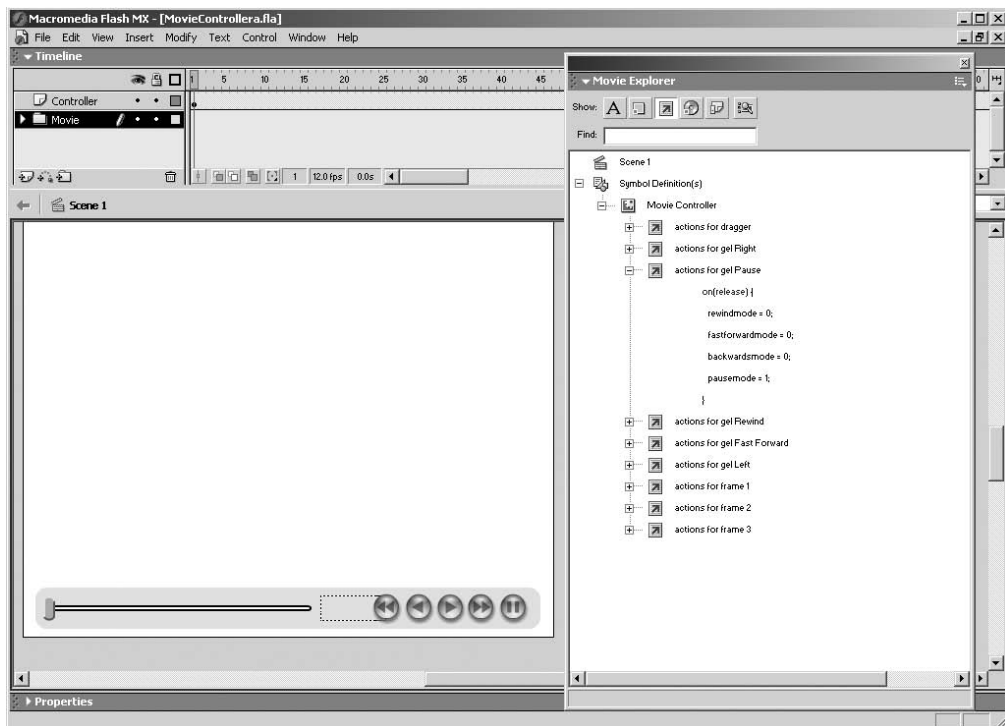


Figure 13.16 The controller is applied through the library

a value of 1 or 0 applied to one of the four modes of play (backwards, rewind, fast forward and pause), which turns that mode on or off.

You can easily change the graphical appearance of the movie controller by updating the graphics that make up the buttons. The ActionScript to drive the play and fast forward buttons is shown below:

```
on(release) {
    rewindmode = 0;
    fastforwardmode = 0;
    backwardsmode = 0;
    pausemode = 0;
}
on(release) {
    rewindmode = 0;
    fastforwardmode = 1;
    backwardsmode = 0;
    pausemode = 0;
}
```

Orbit

The orbit example is a simple way to create complex orbit effects. The example file, Orbit.fla from the website, contains the “sun” with the “earth” orbiting around it, and the “moon” orbiting around the earth. To set up an orbit, first of all drag the “Orbit Generator” onto the stage and open its actions panel. You will see a number of areas that can be changed to affect how the orbit function works; the movie clip that is referenced in the “movclip” field is the movie clip that is being orbited around, and the “movclip2” field is the movie clip that is orbiting. You can then set

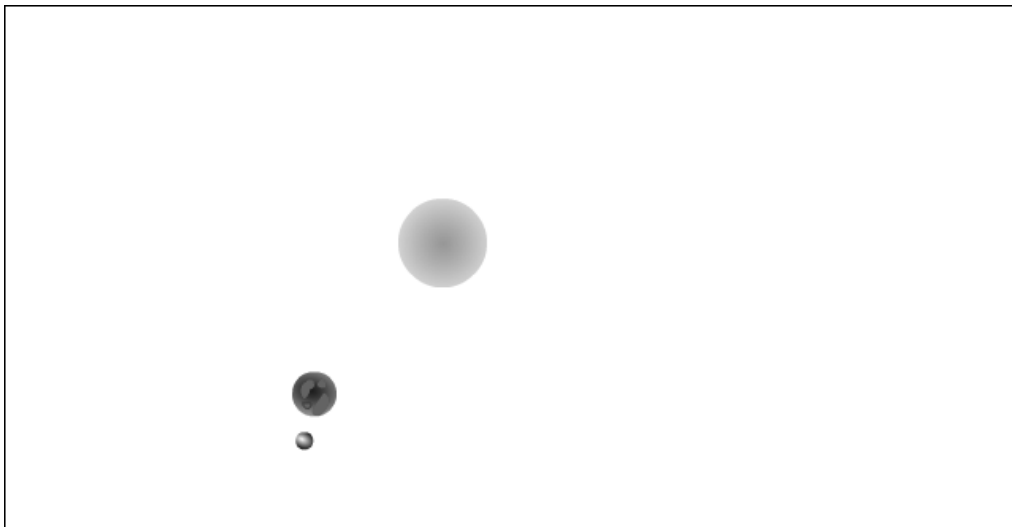


Figure 13.17 *Set objects to orbit around others*

a number of variables in the same actions panel: you can set the angle of the orbit, which is the angle that the orbiting object starts at, the speed of the orbit (the higher the faster) and the radius of the orbit, which sets how far out the orbit begins.

As you can see in the example movie, you can set objects to orbit around others that are moving across the screen, which can lead to some stunning effects, and you can put as many orbits on the screen as you like. The areas of the ActionScript that you can edit are outlined below:

```
// direction = 'ccw';
// The direction of movement           (either cw or ccw)

// radius = 100;
// The radius of the orbit

// speed = 6;
// The speed of the orbit

// angle = 180
// This sets the initial angle         the movie clip starts at:
// 180 - top
// 90 - right
// 270 - left
// 0, 360 - bottom

// movclip = 'earth'
// The name of the movie clip to       orbit around

// movclip2 = 'moon';
// The name of the movie clip to       animate (must be quoted as
//                                     a string)
```

Snow

This handy snow effect can be plugged into any movie simply by dragging the Snow Generator movie clip onto the stage. The snow falls down the screen randomly, and follows a sine wave to give a wavy snowing effect. Check out Snow.fla on the website (www.sprite.net/understanding).

The ActionScript that controls how the snow falls can be updated depending on the movie size and how much snow you want to fall. The actions for the Snow Generator can be found in the first keyframe of the actions layer of the Snow Generator movie clip. You can alter the amount of snow by entering a value for the `amount_of_snow` variable and the `snow_speed` variable controls how fast the snow falls, handy if you want to whip up a snowstorm! The movie dimensions in the ActionScript should be set to the height and width of the movie you are putting the effect into and you can change the sine values to affect how the snow falls. The ActionScript that you can edit looks like this:

```
amount_of_snow = 50; // This sets the amount of snow
snow_speed = 2; // This sets the speed that the snow falls
```

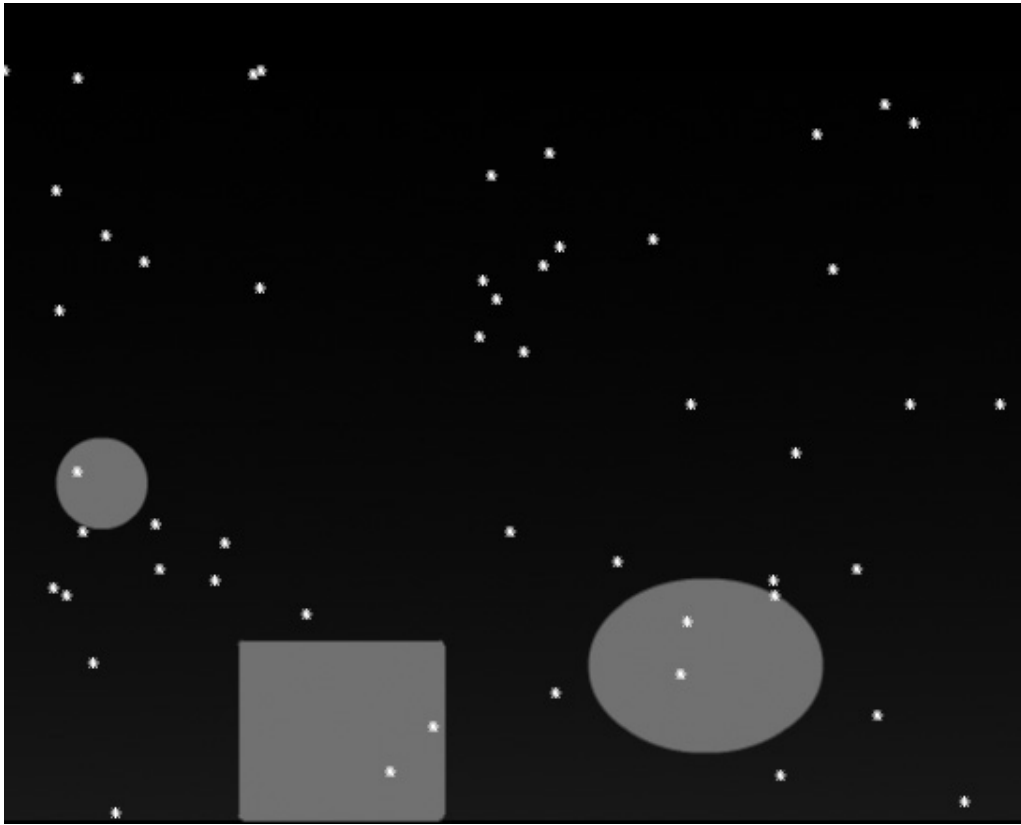
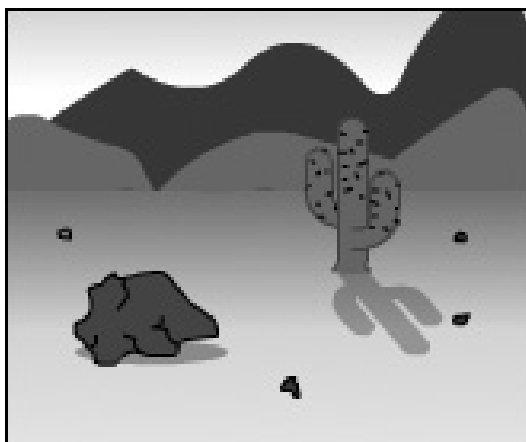


Figure 13.18 *A snowstorm generator*

```
movie_width = 550; // Movie dimensions
movie_height = 400;
sinTable = new Array(movie_height); // Precalc SIN values,
speeds things up a little for (i = -movie_height; i <
movie_height; i++) { sinTable[i] = Math.sin(i/8); }
```

The Ultimate Test

The ultimate test of your ability is to use ActionScript to communicate to your computer what you see in your imagination. Good ideas are never constrained by your technical ability; you will always find a solution to every problem. Hopefully this book has started you on the first steps of exploring how you can stretch the whole Flash environment and take it beyond its very constrained limits. I would welcome an invitation to look at anything this book has helped you produce. Good luck.



**APPENDIX A:
ACTIONSCRIPT
TERMINOLOGY**

**APPENDIX B:
LETTERS A TO Z
AND STANDARD
NUMBERS 0 TO 9**

Appendices

This Page Intentionally Left Blank

Appendix A:

ActionScript Terminology

Like any scripting language, ActionScript uses its own terminology. The following list provides an introduction to important ActionScript terms in alphabetical order.

Actions are statements that instruct a movie to do something while it is playing. For example, `gotoAndStop` sends the playhead to a specific frame or label. In this book, the terms action and statement are interchangeable.

Boolean is a true or false value.

Classes are data types that you can create to define a new type of object. To define a class, you create a constructor function.

Constants are elements that don't change. For example, the constant `Key.TAB` always has the same meaning: it indicates the Tab key on a keyboard. Constants are useful for comparing values.

Constructors are functions that you use to define the properties and methods of a class. For example, the following code creates a new `Circle` class by creating a constructor function called `Circle`:

```
function Circle(x, y, radius){  
    this.x = x;  
    this.y = y;  
    this.radius = radius;  
}
```

Data types are sets of values and the operations that can be performed on them. The ActionScript data types are string, number, boolean, object, movieclip, function, null, and undefined.

Event handlers are special actions that manage events such as `mouseDown` or `load`. There are two kinds of ActionScript event handlers: actions and methods. There are only two event handler actions, `on` and `onClipEvent`. In the Actions toolbox, each ActionScript object that has event handler methods has a subcategory called Events.

Events are actions that occur while a movie is playing. For example, different events are generated when a movie clip loads, the playhead enters a frame, the user clicks a button or movie clip, or the user types at the keyboard.

Expressions are any legal combination of ActionScript symbols that represent a value. An expression consists of operators and operands. For example, in the expression `x + 2`, `x` and `2` are operands and `+` is an operator.

Functions are blocks of reusable code that can be passed parameters and can return a value. For example, the `getProperty` function is passed the name of a property and the instance name of a movie clip, and it returns the value of the property. The `getVersion` function returns the version of the Flash Player currently playing the movie.

Identifiers are names used to indicate a variable, property, object, function, or method. The first character must be a letter, underscore (`_`), or dollar sign (`$`). Each subsequent character must be a letter, number, underscore, or dollar sign. For example, `firstName` is the name of a variable.

Instance names are unique names that allow you to target movie clip and button instances in scripts. You use the Property Inspector to assign instance names to instances on the Stage. For example, a master symbol in the library could be called `counter` and the two instances of that symbol in the movie could have the instance names `scorePlayer1` and `scorePlayer2`. The following code sets a variable called `score` inside each movie clip instance by using instance names:

```
_root.scorePlayer1.score += 1;
_root.scorePlayer2.score -= 1;
```

Instances are objects that belong to a certain class. Each instance of a class contains all the properties and methods of that class. All movie clips are instances with properties (for example, `_alpha` and `_visible`) and methods (for example, `gotoAndPlay` and `getURL`) of the `MovieClip` class.

Keywords are reserved words that have special meaning. For example, `var` is a keyword used to declare local variables. You cannot use a keyword as an identifier. For example, `var` is not a legal variable name.

Methods are functions assigned to an object. After a function is assigned, it can be called as a method of that object. For example, in the following code, `clear` becomes a method of the `controller` object:

```
function reset( ){
    this.x_pos = 0;
    this.x_pos = 0;
```

```

}
controller.clear = reset;
controller.clear( );

```

Objects are collections of properties and methods; each object has its own name and is an instance of a particular class. Built-in objects are predefined in the ActionScript language. For example, the built-in Date object provides information from the system clock.

Operators are terms that calculate a new value from one or more values. For example, the addition (+) operator adds two or more values together to produce a new value. The values that operators manipulate are called operands.

Parameters (also called arguments) are placeholders that let you pass values to functions. For example, the following welcome function uses two values it receives in the parameters `firstName` and `hobby`:

```

function welcome(firstName, hobby) {
    welcomeText = 'Hello, ' + firstName + 'I see you enjoy ' +
hobby;
}

```

Properties are attributes that define an object. For example, `_visible` is a property of all movie clips that defines whether a movie clip is visible or hidden.

Target paths are hierarchical addresses of movie clip instance names, variables, and objects in a movie. You name a movie clip instance in the movie clip Property Inspector. (The main timeline always has the name `_root`.) You can use a target path to direct an action at a movie clip or to get or set the value of a variable. For example, the following statement is the target path to the variable `volume` inside the movie clip `stereoControl`:

```

_root.stereoControl.volume

```

Variables are identifiers that hold values of any data type. Variables can be created, changed, and updated. The values they store can be retrieved for use in scripts. In the following example, the identifiers on the left side of the equal signs are variables:

```

x = 5;
name = 'Lolo';
customer.address = '66 7th Street';

```

Appendix B:

Letters A to Z and Standard Numbers 0 to 9

The following table lists the keys on a standard keyboard for the letters A to Z and the numbers 0 to 9, with the corresponding ASCII key code values that are used to identify the keys in ActionScript.

Letter or number key	Key code
A	65
B	66
C	67
D	68
E	69
F	70
G	71
H	72
I	73
J	74
K	75
L	76
M	77
N	78
O	79
P	80
Q	81
R	82
S	83
T	84
U	85
V	86
W	87
X	88
Y	89
Z	90
0	48
1	49
2	50
3	51
4	52
5	53
6	54
7	55
8	56
9	57

Keys on the Numeric Keypad

The following table lists the keys on a numeric keypad, with the corresponding ASCII key code values that are used to identify the keys in ActionScript.

Numeric keypad key	Key code
Numpad 0	96
Numpad 1	97
Numpad 2	98
Numpad 3	99
Numpad 4	100
Numpad 5	101
Numpad 6	102
Numpad 7	103
Numpad 8	104
Numpad 9	105
Multiply	106
Add	107
Enter	108
Subtract	109
Decimal	110
Divide	111

Function Keys

The following table lists the function keys on a standard keyboard, with the corresponding ASCII key code values that are used to identify the keys in ActionScript.

Function key	Key code
F1	112
F2	113
F3	114
F4	115
F5	116
F6	117
F7	118
F8	119
F9	120
F10	121
F11	122
F12	123
F13	124
F14	125
F15	126

Other Keys

The following table lists keys on a standard keyboard other than letters or numbers, with the corresponding ASCII key code values that are used to identify the keys in ActionScript.

Key	Key code
Backspace	8
Tab	9
Clear	12
Enter	13
Shift	16
Control	17
Alt	18
Caps Lock	20
Esc	27
Spacebar	32
Page Up	33
Page Down	34
End	35
Home	36
Left Arrow	37
Up Arrow	38
Right Arrow	39
Down Arrow	40
Insert	45
Delete	46
Help	47
Num Lock	144
; :	186
= +	187
- _	189
/ ?	191
\Q ~	192
[{	219
\	220
] }	221
" ,	222

Index

Actions Panel:

- auto formatting, 8–9
- descriptions of an Action, viewing, 7
- display procedure, 6
- Expert Mode, 6, 7, 9–10
- features, 5
- line number viewing, 8
- mode switching, 9–10
- Normal Mode, 6, 7, 9–10
- parameters, working with, 7
- preference setting, 14
- printing Actions, 8
- punctuation balance checking, 9
- Reference panel, 10
 - font size selection, 10
 - sample code copying and pasting, 10
- scripts:
 - pinning to the Actions Panel, 8
 - replacing text, 7
 - searching for text, 7–8
- scrolling, 7
- selection procedure, 6–7
- syntax checking, 8
- toolbox area, 6–7
 - resizing, 8

Actions/statements:

- about actions, 247
- duplicateMovieClip(), 39–40
- locations for assignment, 18
- removeMovieClip(), 39
- setProperty(), 45
- startDrag(), 37–9
- stopDrag(), 38
- that target movie clips, 36
- to control movie clips, 35–7
- “with” action for movie clip addressing, 41
- “with” statement and function, 41–3

ActionScript:

- about ActionScript, 3–4
- and ECMA-262, 4
- and JavaScript language, 4
- the language, 5
- syntax *see* Syntax for ActionScript
- terminology, 247–9

ActionScript-driven effects:

- about driven effects, 223–5
 - Animated Bug, 236–7
 - Bubbles, 235–6
 - Change Cursor, 227–8
 - Dancing Lights, 237–8
 - Depth Swapping, 228
 - Drag and Drop on Grid, 225–6
 - Earthquake, 238–9
 - Gravity Footballs, 239–41
 - Magic Sparkle, 233
 - Magnifying Glass, 231–2
 - Movie Controller, 241–2
 - Moving Eyes, 226–7
 - Orbit, 242–3
 - Parallax Scrolling, 229–31
 - Snow, 243–4
 - Sparkle, 232
 - Starfield, 233–5
 - Wobble Button, 228–9
- ## AddListener(listener), 97–8
- ## Animated Bug effect, 236–7
- ## Appearance properties, 45–6
- ## Arguments, and functions, 72
- ## Arrays:
- about arrays, 79–80
 - array access operators, 65–6
 - array elements, 80
 - adding and inserting, 84–6
 - array objects, 80–2, 83–4, 85
 - and assignment operator (=), 83
 - concat() method, 83
 - creation, 81
 - join() method, 84
 - length, 82–3
 - manipulation methods, 87
 - methods summary, 85
 - push() method, 84–6
 - retrieving elements from, 81
 - size, 80
 - slice() method, 84
 - sort() method, 87
 - sortOn() method, 87
 - toString() method, 87

Arrays (*cont'd*)

- trace() method, 87

- Arrow keys, moving objects with, 100–2

- ASCII key code values, 250–2

- Assets defining, source movies, 27–8

- Assets sharing, 26–7

- assets definition, 27–8

- author-time shared assets, 27

- conflict resolution, 33–4

- destination movies, 28–30

- for font management, 30–2

- Linkage Properties box, 31

- linking from a destination movie, 28–30

- name conflicts, 34

- runtime shared assets, 26, 27

- Assignment operator (=), 53, 63, 64–5, 83

- Assignment operators, list of all, 65

- Associativity of operators, 60

- AttachMovie action, 139–40

- AttachMovie method, 46–8

- AttachSound method, 46–8

- Attributes, with XML, 148–9, 161–2

- Author-time sharing facilities for updating/replacing symbols, 30

- Auto formatting, 8–9

- Bandwidth Profiler, 218–19

- Bitwise operators, 64, 131–2

- Boolean, 247

- Bubbles effect, 235–6

- Built-in functions, 78–9

- Buttons:

- about buttons, 15

- button events, 35

- Button Instance, basic usage, 3

- creation of, 15–16

- with movie clips, 21–3

- Up, Over, Down and Hit states, 15–16

- Centralizing code for functions, 79

- Change Handlers, 168

- CheckBox components, 168–9, 178, 181–2

- CheckBox properties, 169

- CheckMousePosition(), 136, 139

- Child/grandchild concept, 151–2, 157–61

- Classes, about classes, 247

- Code driven animation, 223–4

- Code hints/hinting, 12–14, 54

- Color object:

- about Color object, 125

- attachMovie action, 139–40

- bitwise operators, 131–2

- CheckMousePosition(), 136, 139

- color tints, 126

- creating new, 125–6

- dragColor() function, 130

- draw methods, 142–5

- drawing tools, 139

- dropColor() function, 130–1

- hitTest function, 133, 137

- Identifier name text table, 140

- method summary, 126

- SetCurrentColor(), 136

- ShapeCount, 140–1

- tutorials:

- making part of a color application, 126–9

- with Property Inspector, Advanced mode, 126, 127

- sliders for color adjustment, 136–42

- with ZakPainter, 133–6

- virtual “pens”, 142–3

- ZakPainter, 133–6, 145

- tool buttons scripting, 142

- Colored syntax, 94

- ComboBox component, 169–71, 178, 180–1

- Comments, 94

- Comparison operators, 61–2

- Components:

- about components, 165

- about Flash professional version of, 165

- adding components, 166

- adding to Flash documents, 168, 177–80

- Change Handlers, 168

- CheckBox, 168–9, 178, 181–2

- ComboBox, 169–71, 178, 180–1

- components panel, 165–6

- Global Skins folder, 166

- ListBox, 171–2, 178–9, 182

- Live Preview, 167

- Parameters Panel, 167

- with Property Inspector, 167

- PushButton, 172–4, 180, 183

- RadioButton, 174, 178, 181

- ScrollBar, 175–6

- with text fields, 176

- ScrollPane, 176–7, 179–80, 182–3

- tutorials:

- adding components, 177–80

- configuring components, 180–3

- entire movie, 183–6

- movie testing, 180

- overview, 177
 - viewing, navigation demo, 177
- UI Components folder, 166
- Concat() new array method, 83
- Conditional statements:
 - about conditional statements, 66
 - conditional operator, 69–70
 - “for” statement/loop, 68–9
 - “if” statement, 66–7
 - “while” statement/loop, 67
- Constants, about constants, 95, 247
- Constructors, about constructors, 247
- Curly braces, 92
- Cursor-Change of Cursor, 227–8
- Dancing Lights effect, 237–8
- Data, and functions, 76
- Data types, about data types, 247
- Debugger, with movies, 23
- Declarations, 54–5, 73
 - XML, 147
- Depth Swapping, 228
- Destination movies, linking shared assets, 28–30
- Dot operators, 65
- Dot syntax, 18–19
- Drag and Drop on Grid, 225–6
- DragColor() function, 130
- Drawing tools, with Color object, 139
- DropColor() function, 130–1
- duplicateMovieClip() action, 39–40
- Earthquake effect, 238–9
- ECMA-262 (European Computer Manufacturers Association-262), 4, 91
- Empty movie clips:
 - about empty movie clips, 48
 - dynamic creation of, 48
 - EmptyMovieClip method, 48
- Equality operators, 53, 62–3
 - and assignment operator (=), 53, 63
 - equality (==), 53, 62
 - strict equality (===), 63
- Errors in syntax, highlighting, 12
- Events and event handlers:
 - about events, 35, 247
 - button events, 35
 - movie clip events, 35
- Expert Mode, 6, 7
 - mode switching, 9–10
- Exporting movies *see* Publish command
- Expressions, 248
- External text editors, code and scripts:
 - adding an external script to exporting movies, 11–12
 - externalizing ActionScript code, 11
 - importing text files, 11
 - use of, 11
- Flash MX *see* Macromedia Flash
- Flash timeline *see* Timelines
- Flash UI Components *see* Components
- Font management, with shared library assets, 30–2
- “For” statement/loop, 68–9
- Frames:
 - frame actions, 16
 - jumping to, 16–18
- Fscommand(), 57–8
- Functions:
 - about functions, 72–4, 248
 - and arguments, 72
 - built-in functions, 78–9
 - calling functions, 72, 73
 - centralizing code for, 79
 - and data, 76
 - and declarations, 73
 - function scope, 78
 - functions within functions, 76–7
 - with parameters, 74–5
 - returning values from, 75–6
 - subroutines, 76
 - value passing, 77–8
- Games:
 - assets identification, 204–5
 - assets preparation, 205–6
 - finishing a game design, 221–2
 - game ending, 204
 - game loop, 203–4
 - game updating, 203
 - introductions, 203
 - player inputs, 203
 - properties with assets, identification, 205
 - screen displaying, 204
 - textual descriptions, 204
 - see also* Goose Attack game; Publish command; Trivia Quiz game
- GetAscii() method, 96–7
- getCode() method, 96–7, 103–4
- Global Skins folder, 166
- Go To action, 16–18
- Goose Attack game:
 - about Goose Attack, 207

- Goose Attack game (*cont'd*)
 - assets preparation, 205–6
 - elapsed time, 210
 - game start, 211
 - geese generation, 208–10, 213
 - geese moving, 211–14
 - opening screen, 207–9
 - target:
 - addition of, 214–16
 - game over screen, 217–18
- Gravity Footballs, 239–41
- Hints, code hints and tooltip-style hints, 12–14, 54
- HitTest function, 133, 137
- Hungarian Notation, 54
- Identifiers:
 - about identifiers, 248
 - name text table, 140
- “If” statement, 66–7
- Images, loading dynamically, loadMovieNum
 - method, 44–5
- Incremental operators, 60–1
- Instances/instance names, about instances, 99–100, 248
- “Invocation”, 73
- isDown() method, 96–7
- isToggled(keycode), 97
- JavaScript:
 - about JavaScript, 4
 - and Fscommand(), 57
- Join() method for arrays, 84
- Jumping to a frame or scene, 16–18
- Key object:
 - about key object, 96–7
 - addListener(listener), 97–8
 - getAscii() method, 96–7
 - getCode() method, 96–7, 103–4
 - isDown() method, 96–7
 - isToggled(keycode), 97
 - listener object, 102–3
 - moving a MovieClip, 104–5
 - property summary, 98
 - removeListener(listener), 98
- Key press/release detection, 99–100
- combinations detection, 107–8
- disableKeyDetect() function, 107
- enableKeyDetect() function, 107
- keyDetect() function, 107
- Keyboard input capture, 98–9
- Keyframes:
 - basic usage, 3
 - with components, 183
- Keywords:
 - about keywords, 91, 248
 - case sensitivity/capitalization, 91
 - list of, 95
 - “this” with Sound objects, 117
- Layers, in Trivia Quiz game, 187, 194
- Level numbers, movie loading, 20
- Library assets sharing *see* Assets sharing
- Line number viewing, 8
- Linkage Properties box, 31
- ListBox components, 171–2, 178–9, 182
- Live Preview, with components, 167
- Loading images or an SWF dynamically,
 - loadMovieNum method, 44–5
- Loading sounds dynamically, loadsound(URL, isStreaming) method, 43–4
- Logical operators, 63–4
- Macromedia Flash:
 - Flash MX professional:
 - features, 3, 165
 - main splash screen, 4
 - standard and professional versions, 3
 - versions 3, 4 and 5, features, 4
- Magic Sparkle effect, 233
- Magnifying glass fun tool, 231–2
- Manipulation methods, arrays, 87
- Menu-style hints, 13
- Methods, about methods, 248–9
- Methods to control movie clips, 35–7
- Modes *see* Expert Mode; Normal Mode
- Modes:
 - Expert Mode, 6, 7, 9–10
 - Normal Mode, 6, 7, 9–10
- Movie Controller (dynamic control of movies), 241–2
- Movie Explorer, 14–15
- Movies/movie clips:
 - about movie clips, 21–3
 - actions to control movie clips, 35–7
 - adding dynamically to a movie from library, 46–8
 - assets defining, source movies, 27–8
 - attaching to another movie clip, 47–8
 - author-time sharing facilities, 30
 - with buttons, 21–3
 - changing position and appearance, 45–6

- controlling, 21
 - Movie Controller, 241–2
- debugger display, 23
- empty movie clip creation, 48
- example, 24–6
- hierarchy of clips, 24
- instances, 99–100, 248
- level stacking order, 23
- load action, 20
 - with level numbers, 20
- methods to control movie clips, 35–7
- movie clip events, 35
- moving using Key Object, 104–5
- naming, 21–2, 47
- nested movies, 21
- objects methods, 35
- preloaders, 49–50
- replacing symbols, 30
- scriptable masks, 49
- source movies, asset defining, 27–8
- testing actions, 21
- unload action, 21
- updating symbols, 30
- see also* Assets sharing; Timelines
- Moving Eyes, 226–7
- Moving objects with the arrow keys, 100–2
- MP3 (external), loading dynamically into Sound object, 123–4
 - common error messages, 124
- Normal Mode, 6, 7
 - mode switching, 9–10
- Numeric operators, 61
- Object-Oriented Programming (OOP), 95–6
- Objects:
 - about objects, 249
 - object classes, suffixes for, 14
 - see also* Color object; Key object; Sound object
- OnSoundComplete method, 122–3
- OOP (Object-Oriented Programming), 95–6
- Operators/operands:
 - about operators, 58, 249
 - ActionScript operator list, 59–60
 - array access operators, 65–6
 - assignment operator (=), 53, 63, 64–5, 83
 - assignment operators, list of all, 65
 - associativity, 60
 - bitwise operators, 64
 - comparison operators, 61–2, 62
 - dot operators, 65
 - equality operators, 62–3
 - incremental operators, 60–1
 - logical operators, 63–4
 - numeric operators list, 61
 - precedence, 58, 60
 - string operators, 62
- Orbit effect, 242–3
- Panning sound, 117
- Parallax Scrolling, 229–31
- Parameters:
 - about parameters, 249
 - Parameters Panel, 167
 - working with, 7
- Parent syntax/movies, 19
- Parentheses, 93
- Precedence of operators, 58, 60
- Preferences setting, Actions panel, 14
- Preloaders, 49–50
- Professional Flash MX, 3, 4, 165
- Properties:
 - about properties, 249
 - appearance properties, 45
 - read-only properties, 45–6
 - setProperty() action, 45–6
- Property Inspector:
 - Advanced mode, 126, 127
 - with components, 167, 180–3
- Publish command:
 - about Publish command, 218
 - optimization, 218–20
 - Publish Settings, 220–1
- Punctuation balance checking, 9
- Push() method for arrays, 84–6
- PushButton component, 172–4, 180, 183
- RadioButton component, 174, 178, 181
 - in Trivia Quiz game, 192–3
- Read-only properties, 45–6
- Reference panel:
 - font size selection, 10
 - sample code copying and pasting, 10
 - using, 10
- removeListener(listener), 98
- removeMovieClip() action, 39
- Return statements, 74–5, 76
- Runtime shared assets, 26, 27
- Scenes, jumping to, 16–18
- Scriptable masks, 49

- Scrolling:
 - in Action Panel, 7
 - Parallax Scrolling, 229–31
 - ScrollBar component, 175–6
 - in Trivia Quiz game, 188
 - ScrollPane component, 176–7, 179–80, 182–3
- Semicolons, 92–3
- SetColor(), 136
- setInterval() method, 105
- setProperty action, 45
- ShapeCount, 140–1
- Shared ActionScript, 32–3
- Shared library assets *see* Assets sharing
- Slice() method for arrays, 84
- Snow effect, 243–4
- Sound object:
 - about sound object, 109–11
 - continue/play button, 121–2, 123
 - ending, and actions after, 122–3
 - event handler summary, 110
 - independent Sound objects, control of, 116–18
 - “loop”, 115
 - method summary, 109
 - MP3 (external), loading dynamically, 123–4
 - multiple sound handling, 111
 - naming suggestions, 114–15
 - “offset”, 115
 - onSoundComplete method, 122–3
 - panning, 117
 - pause button, 121–2
 - play/continue button, 121–2, 123
 - property summary, 110
 - starting, 115
 - stop button, 121–2, 123
 - stopping, 115
 - tutorials, two button control, 111–14
 - volume control, 118–19
 - volume slider, build of, 119–21
- Sounds, loading dynamically:
 - attachsound method, 46–8
 - loadsound(URL, isStreaming) method, 43–4
- Source movies, asset definition, 27–8
- Sparkle effect, 232
- Starfield effect, 233–5
- StartDrag() action, 37–9
- Statements:
 - and Button Instance and keyframe, 3
 - return statements, 74–5
 - switch statement, 70–1
 - see also* Actions/statements; Conditional statements
- StopDrag(), 38
- Strict equality (===) operator, 63
- String operators, 62
- Subroutines, 76
- Suffixes, supported suffixes list, 54
- SWF, loading dynamically, loadMovieNum method, 44–5
- Switch statement, 70–1
- Symbols, updating or replacing, 30
- Syntax for ActionScript, 92–5
 - checking, 8
 - colored syntax, 94
 - comments, 94
 - constants, 95
 - curly braces, 92
 - error highlighting, 12
 - keywords, 95
 - parentheses, 93
 - semicolons, 92–3
- Tags, in XML, 147–8
- Target paths, about target paths, 249
- tellTarget method, 42–3
- Terminology of ActionScript, 247–9
- Text editors *see* External text editors
- Text files, importing and exporting, 11
- “this” keyword, 117
- Timelines:
 - absolute paths, 24, 25
 - with ActionScript-driven effects, 223
 - controlling and target, 24
 - multiple, 23–4
 - nested, 24
 - relative target paths, 24, 25–6
- Toolbox, Actions Panel, 6–7, 8
- Tooltip-style hints, 12–14
- ToString() method for arrays, 87
- Trace() method for conversion to string values, 87
- Trivia Quiz game:
 - about the game, 187
 - background, 187
 - end of game code, 198–9
 - finishing touches, 197–9
 - layers, 187, 194
 - questions display, 192–7
 - questions XML file, 189–91
 - running requirements, 196–7
 - start quiz button, 190–2
 - structure, 187–8
 - title page, 187–8

- UI Components *see* Components
- updateAfterEvent() method, 105–6
- URLs (Universal Resource Locators):
 - and LoadVars, 56–7
 - and shared assets, 28–9
 - as variables, 53
- Value passing, and functions, 77–8
- Variables:
 - about variables, 53–5, 249
 - assignment operator (=), 53, 63, 64–5, 83
 - code hinting, 54
 - composition, 53
 - creation, 53, 54
 - declaring, 54–5
 - external variables loading, 56–8
 - Fscommand(), 57–8
 - LoadVariables(), 57
 - LoadVars(), 56–7
 - local, 55–6
 - _root, 55
 - scope, 55
 - suffixes, 54
 - supported list, 54
 - value changing, 55–6
 - see also* Operators/operands
- Virtual “pens”, 142–3
- “While” statement/loop, 67
- “With” action for movie clip addressing, 41
- “With” statement and function, 41–3
- Wobble Button, 228–9
- XML (extensible markup language):
 - about XML, 146
 - attributes, 148–9, 161–2
 - benefits, 146
 - child/grandchild concept, 151–2, 157–61
 - the declaration, 147
 - Document Type Definitions (DTDs), 149
 - for exchange and storage of data, 147
 - HTML differences, 146
 - parser, 149
 - pointers, 158
 - status property values, 150–1
 - status property values list, 150–1
 - tags, 147–8
 - in Trivia Quiz game, 189–92
 - tutorials:
 - data from XML within Flash, 155–61
 - loading into Flash, 153–5
 - why and when to use, 152–3
 - XML Document Object Model (DOM), 147
 - XML objects, 150–1
 - XML tree structure, 151
 - XMLnode class, 151–2
 - XMLSocket object, 162
- ZakPainter, 133–6
 - tool buttons scripting, 142