CHAPTER 1

ACTIONSCRIPT OVERVIEW

While you likely know what ActionScript is and are eager to begin working with the new version, a brief overview of its development will give you some insight into its use—particularly related to Flash Player and how it handles different versions of ActionScript. This brief introductory chapter will give you a quick look at where ActionScript 3.0 fits into your workflow, and will cover:

- **What Is ActionScript 3.0?** It's to be expected that a new version of ActionScript will bring with it new features. However, this version has been written anew from the ground up and is even handled separately from previous versions of ActionScript at runtime. This intentional splintering of Flash Player affords significant performance increases, but also brings with it limitations as to how multiple versions of ActionScript interact.
- **The Flash Platform.** At the time of this writing, ActionScript 3.0 is the internal programming language of Flex and AIR (the Adobe Integrated Runtime application). Differences in compiling and environment-specific attributes prevent every file written in ActionScript 3.0 from working in every aspect of the Flash Platform, but the fundamentals—indeed the bulk—of the language is the same throughout.
- **Procedural Versus Object-Oriented Programming.** A great deal of attention has been focused on the object-oriented programming (OOP) capabilities of ActionScript 3.0, and the power and robustness of the language really shine in this area. However, you'll be happy to learn that a move to ActionScript 3.0 doesn't mean that you must become an expert at OOP. It is still possible to use a structured collection of functions, which characterize procedural programming, to author ActionScript 3.0 projects. In addition, using Flash CS3, it is still possible to code in the timeline, rather than coding exclusively with external classes. If you prefer object-oriented programming, enhancements to ActionScript 3.0's OOP infrastructure make it more robust and bring it more in line with the features of other important, OOP-based languages (such as Java) and make moving between such languages a bit easier.

IN THIS CHAPTER

- **The `Document` Class.** Object-oriented programming is not for everyone, but for those starting on this journey, Flash CS3 offers a simpler entrance to an OOP application by way of the `Document` class. An attribute of the Properties Inspector, you need only specify which external class is your starting point, and no timeline script is required.
- **Legacy Code Compatibility.** Because ActionScript 3.0 cannot co-mingle with previous versions of the language in the same file, developing projects that support older code is a chllenge. We'll briefly introduce the issues involved, and discuss them in greater depth in a later chapter.

What Is ActionScript 3.0?

Although the new version of Flash's internal scripting language contains much that will be familiar to users of prior versions, it's probably best to think of ActionScript 3.0 as entirely new, for a few simple reasons. First, a few things are quite different, such as the event model and the way assets are displayed. Second, subtle changes run throughout the language and require some attention until they become second nature. These are usually small concerns, such as a slight change in the name of a property.

Most importantly, however, ActionScript 3.0 has been rewritten from the ground up and uses a different code base than prior versions of the language. This optimization provides relatively dramatic performance increases, but it means that ActionScript 3.0 code cannot be mixed with prior versions of the language in the same file.

The newness of this version, however, shouldn't intimidate you. It's true that the learning curve for ActionScript 3.0 is steeper than for prior versions, but that is usually a function of its robustness more than one of difficulty. Typically, there is an adjustment period during which users must occasionally adapt to a slightly new way of doing things.

To help you get over any possible trepidation, here's a look at some of the highlights of the new features of ActionScript 3.0. Keeping these benefits in mind may help make it easier to accept change, particularly when that change may initially seem tedious or overly complicated. Select new features include:

More detailed error reporting

ActionScript 3.0 requires strict data typing of variables, arguments, function returns, and so on. This data typing is discussed in Chapter 2, but boils down to telling the compiler what kind of data you expect to be working with at any specific time. Data type checking was introduced in ActionScript 2.0 but was previously optional. The heightened data typing enforcement improves error checking and provides more information while coding to allow you to correct the problem. Further, ActionScript 3.0 now

enforces static data typing at runtime. This improves data type reliability at runtime, and also improves performance and reduces memory usage because the data types are stored in machine code rather than having to be dynamically addressed at runtime.

Syntax improvements

Syntax issues have been unified and cleaned up throughout the language. For example, property names have been clarified in some cases, and have been made consistent by removing the occasional leading underscores, as you'll see in Chapter 3. Also, multiple, subtly different ways of approaching the same or similar tasks have been made consistent, such as when loading external assets (discussed in Chapter 13) or linking to a URL (as seen throughout the book).

New display architecture

The many previous methods to dynamically add something to the display environment are now consolidated. The new display list simplifies this process significantly and also makes it easier to change the visual stacking order, as well as parent, child, and sibling hierarchical relationships, of display objects. As a major change introduced by ActionScript 3.0, we discuss this at length in Chapter 4.

New event architecture

Still another example of improved consistency, all events are now fielded by event listeners—essentially listening for a specific event to occur, and then reacting accordingly. The new event model is also more powerful, allowing mouse and keyboard events to propagate through multiple objects in the display list. The event model is discussed in Chapter 3.

Improved XML handling

A formerly cumbersome process, working with complex XML documents is now a pleasure with ActionScript 3.0. Adopting the standard commonly referred to as E4X, ActionScript now treats XML objects in a much more intelligent and familiar manner. The new approach allows you to use the same dot syntax to string related objects together.

More text scripting options

New text-processing methods now allow for much finer control over text manipulation. You can now find the text of a particular line in a text field, the number of characters in that line, and the character at a specified point (such as under the mouse). You can also find the index in the text field of the first character in a paragraph, and even get the minimum-bounding rectangle surrounding any specific character. All these options not only make working with a text field easier, but also allow a tighter integration with the lines and characters in a field and their surrounding stage elements. Text is discussed in Chapter 10.

New regular expressions

Another boon to text handling is the new native support for regular expressions. Regular expressions are like text manipulation on steroids. Instead of manipulating only specific, known strings of characters, you can now manipulate text using wild cards, character types (numeric, alpha, punctuation, and so on), white space (spaces, tabs, returns), repeating characters, and more. A simple example of regular expression use can be found in Chapter 10.

More sound management options

ActionScript 3.0's new sound capabilities are among the most eye-catching changes to the language. On a practical level, they improve access to both individual sounds and to all sounds playing. Sounds are now placed into separate channels, making it easier to work with multiple individual sounds, but also funnel all sounds through a sound mixer for collective control. You can also now get the amplitude and frequency spectrum data from sounds during playback. Sound is discussed in Chapter 11.

New access to raw data

For more advanced needs, you can now access raw binary data at runtime. Individual bytes of data can be read during download, during sound playback, or during bitmap data manipulation, to name a few examples. These bytes can be stored in a large list and still be accessed quickly and efficiently. We'll show one example of this technique in Chapter 11 when discussing sound visualization.

New automatic scope management

In a programming language, the word *scope* is sometimes used to define the realm in which an object lives. A Flash asset, such as a movie clip, might be in one part of the Flash movie but not another. For example, a child movie clip might be nested inside one of two movie clips found in the main timeline. That nested movie clip exists within one clip but not the other. Its scope, therefore, is restricted to its parent. Programming structures have limited scope, as well, and the challenge is making sure you work within the correct scope when addressing those structures. ActionScript 3.0 greatly simplifies this by automatically tracking scope as you program.

Improved object-oriented programming

Object-oriented programming structures have also been improved in ActionScript 3.0 with the inclusion of sealed classes and new namespaces, among other things. We'll discuss aspects of OOP in this chapter, as well as in Chapter 6, and provide class-based examples throughout the book. New in ActionScript 3.0, all classes are sealed by default, allowing only those properties and methods defined at author time to exist in the class

at runtime. If you do find the need to change classes at runtime—by adding properties, for example—you can still do so by making the classes dynamic. Additionally, namespaces, including the ability to define custom namespaces, allow finer control over classes and XML manipulation.

The Flash Platform

It's important to note that this book focuses primarily on developing ActionScript 3.0 applications using the Flash CS3 Professional integrated development environment (IDE). However, ActionScript 3.0 is the programming language for other Flash Platform applications, as well—notably Flex and AIR (the Adobe Integrated Runtime desktop delivery application).

This means that the scripting skills you develop in Flash CS3 will be largely applicable in other areas of the Flash Platform, extending your reach as a programmer. There are, however, some important differences to understand when examining the big picture of cross-application scripting. We'll give you just a few brief examples here to consider.

To start with, Flash and Flex have different compilers so there is no guarantee that your project will compile correctly in both applications. You can use Flex Builder (the Flex compiler) to compile code-only ActionScript SWFs without the Flex framework, and load them into Flash CS3-generated projects. You can also load Flash CS3-compiled SWFs into a Flex project. However, as soon as you depart from core language needs, things start to get sticky.

For example, Flex does not have the resources of the Flash IDE to create visual assets (such as movie clips) and, by the same token, Flash does not support the Embed tag used by Flex to include such assets. This means that the same code cannot always be used seamlessly when such custom visuals are required. Similarly, the component architecture is different, including a different format and a component set that do not match.

This issue with visual assets has been a hotly debated issue for a while, and progress is being made to smooth the waters a bit. Adobe released a patch for Flex 2, and Flex 3 is in public testing at the time of this writing, improving the compatibility of components. However, it will probably be a while before moving code to and from these applications will be a comfortable process, if it ever happens. At a brisker pace, however, AIR development is becoming more of a crossover affair. Adobe is continuing to work on AIR authoring workflows that originate in Flash CS3.

The thing to keep in mind is that the ActionScript 3.0 language skills you develop will ease your move between applications in the Flash Platform, even if you must work with different authoring tools or compilers to end up with a finished product.

NOTE

AIR projects can also include HTML, JavaScript, and PDF, but ActionScript 3.0 is a large part of its appeal and the language most relevant to this discussion.

Procedural Versus Object-Oriented Programming

Much discussion has been made over the pros and cons of procedural versus object-oriented programming. To touch briefly on this, here is a little background concerning the evolution of ActionScript. ActionScript started as a *sequential* programming language, meaning that scripting was limited to a linear sequence of instructions telling Flash what to do in a step-by-step manner. This approach to scripting was not terribly flexible and did not promote reuse.

As the language evolved, it became a *procedural* programming language. Like sequential programming, procedural programming relied on a step-by-step set of instructions but introduced a more structured, modular approach to scripting. *Procedures*, otherwise known as functions (or, sometimes, subroutines), could be executed again and again as needed from different parts of a project, without copying and pasting copies of the code into the ongoing sequence of instructions. This modularity promoted reuse, and made the code easier to edit and more efficient.

Scripters in search of an even greater degree of modularity and reuse gravitated toward *object-oriented* programming. OOP languages create programs that are a collection of objects. Objects are individual instances of *classes*—collections of code that are self-contained and do not materially alter or disrupt each other. Dividing code into small capsules, appropriately known as *encapsulation*, is one of the hallmarks of an OOP language. Another important feature of OOP is *inheritance*, or the ability to derive classes from parent classes, passing on specific characteristics from the parent.

A classic example of OOP structure, and specifically inheritance, defines a set of transportation vehicles. You might start with a generic `Vehicle` class that includes traits common to all vehicles, such as the basic physics of movement. You might then create three subclasses: `GroundVehicle`, `WaterVehicle`, and `AirVehicle`. These classes would alter or introduce traits specific to ground, water, and air travel, respectively, but not yet be complete enough to represent an actual vehicle. Further derived classes might be `Car` and `Motorcycle` (descending from `GroundVehicle`), `Boat`, and `Submarine` (descending from `WaterVehicle`), and `Plane` and `Helicopter` (descending from `AirVehicle`). Depending on the complexity of your system, you can carry on this process, creating individual models with individual settings for fuel consumption, friction, and so on.

As you can probably imagine, this approach to development adds additional power, flexibility, and prospects for reuse. These benefits, among others, sometimes position object-oriented programming as the best approach to a problem. However, there is a tendency among some programmers to believe that OOP is the best solution to all problems or, effectively, the only solution. This is a faulty assumption.

OOP is often best for very large projects, or for working with a team of programmers, but it can often be overkill for small projects. Additionally, for the uninitiated, it can significantly increase the learning curve, and distract from key topical concepts during your studies. In short, OOP is not always the best tool for the job. Procedural programming still has its place, and Flash CS3 allows you to explore and employ both programming paradigms.

This book attempts to introduce material using both procedural and OOP where appropriate. Using object-oriented practices is a fine goal, and one that we will encourage in this volume. However, we will try first to focus on the material central to each chapter, highlighting syntax and explaining how and why each topic should be addressed in code.

In general terms, we will focus on procedural programming prior to Chapter 6; this chapter serves as a transition chapter between procedural and OOP practices. After Chapter 6, the beginning of each chapter will focus on the topics being discussed, without intrusion by the surrounding OOP class structures. When appropriate, however, each chapter will end with an applied OOP example.

This is our preferred approach to presenting material for all possible users—in both procedural and OOP formats. It is our hope that, regardless of your skill and experience, you will home in on the topics at hand, and then work in the timeline, or in classes, based on your comfort level.

The Document Class

If you decide you would like to start thinking in OOP terms right away, we will show you how to easily take a step in that direction. Flash CS3 introduced a new feature that simplifies associating a main class, or application entry point with your FLA. It is called the *document class* and it does all the work of instantiating the class for you. This means you don't need any code in the timeline at all, and can edit all examples in Flash or the external text editor or development environment of your choice.

Let's start with a simulated chapter example that you might use in the timeline. It does nothing more than use the **`trace()`** method to place a word into the fOutput panel—an authoring-only panel that accepts text output from your file.

```
trace("Flash");
```

To create a document class, you're going to create a kind of wrapper that encloses the **`trace()`** method in the correct class syntax.

NOTE

If you don't plan to start using OOP until we roll it out in later chapters, feel free to skip this section as it will be repeated in Chapter 6. We will provide minimal explanation here just to get you going using the document class, and will explain this material in greater detail in later chapters throughout the book.

Create a new ActionScript file (rather than a new FLA document) and type the following document class shell:

```
1   package {
2
3       import flash.display.MovieClip;
4
5       public class Main extends MovieClip {
6
7           public function Main() {
8
9           }
10
11      }
12  }
```

The first line, along with the closing brace in line 12, defines the class's *package*. A package is a mandatory structure that ensures your class is known to the compiler. Next, you must import any classes that you need to use in your package.

A document class essentially serves as a shortcut for creating an instance of a movie clip or sprite (a new Flash object that is nothing more than a one-frame movie clip) and adding it to the display list so it can be displayed by Flash Player. (This is true even when there is nothing to display, as in this case. We will cover manipulating the display list in Chapter 4.)

All document classes must be derived from either the **`MovieClip`** or **`Sprite`** class. (Other custom classes that are not document classes do not need to be extended from **`MovieClip`** or **`Sprite`** if that is not appropriate.) This example uses **`MovieClip`** so you must import the **`MovieClip`** class, as seen in line 3.

Line 5, along with its closing brace on line 11, is the class definition. Its name is arbitrary but, when naming it, you should follow a few basic rules and conventions. The name should be one word that does not already exist in ActionScript, it should start with an alpha character (rather than a number or other character), and it is typically capitalized. The class must be public, meaning that other classes can access the constructor, and it must extend **`MovieClip`** or **`Sprite`**, as described previously.

Line 7, along with its closing brace on line 9, is the class *constructor*. This is the main function that automatically runs when creating an instance of this class. It, too, must be public and must have the same name as the class. Other functions (if any) can, and must, have unique names. All that remains is to add the lone method required in this case. The constructor must trace "Flash" to the Output panel, so add the following to line 8:

```
7           public function Main() {
8               trace("Flash");
9           }
```

Once finished, you must save the file in the same directory as your FLA file for now. (Later on, you'll learn how to place your class files in other locations.) You must give the file the same name as the class, but add an *.as* extension.

Therefore, this file should be named *Main.as*. Now create a new FLA file, choosing ActionScript 3.0 as its programming language version, and save it in the same directory as your previously created class file. The name of the FLA is unimportant.

Finally, open the Properties Inspector and add the name of your document class, not the name of the document itself, in the Document Class field. In this case, type `Main` instead of `Main.as`, as seen in Figure 1-1.

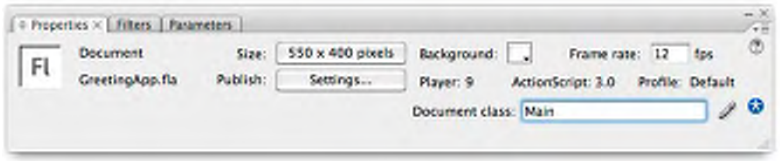

Figure 1-1. Adding a document class to your FLA

Now preview your file. Doing so will create an instance of the `Main` class (which extends `MovieClip` and, therefore, behaves like a movie clip) and add it to the display list. The class will trace "Flash" to the output panel, and your test application will be complete.

Hereafter, you can try any of our timeline code in a document class of your own. Initially, you probably won't know which classes to import or how to make any possible changes to variables or similar structures to conform to the class syntax. However, all the sample code will come with an accompanying class file for testing. You can use those files whenever you wish until you get used to the document class format.

Legacy Code Compatibility

I'd like to end this chapter with a small caveat. You cannot mix ActionScript 1.0 or 2.0 code with ActionScript 3.0 code in the same SWF. You are unlikely to do this if you're learning from scratch, but you may run into this situation if you attempt to update legacy projects by adding ActionScript 3.0 code.

If you ever have the need to run a discrete mixture of ActionScript 3.0 and a prior version of the language, such as showing a legacy file within a new demo interface shell, you can do so by loading a SWF. An ActionScript 3.0 file can load a SWF created in ActionScript 1.0 or 2.0, but it cannot access the older SWF's variables or functions. For all intents and purposes, the same is not true in reverse. An older SWF cannot load an ActionScript 3.0 SWF.

In Chapter 13, we will discuss how to communicate between these two discrete SWFs using a special process. For now, however, just remind yourself again that you cannot combine ActionScript 3.0 with older versions of the language in the same file.

What's Next?

Now that you know a little more about ActionScript 3.0 and the Flash Platform, it's time for a look at some of the fundamentals of the language. By reviewing version-independent concepts at the outset, we can focus on new syntax in subsequent chapters. If you have a lot of experience with ActionScript 1.0 or 2.0, you may wish to skim this material.

In the next chapter, we'll discuss:

- Basic concepts to bring you up to speed quickly, including using the `trace()` method as a diagnostic tool to see immediate feedback from your scripts
- Using variables to store data, including arrays and custom objects that allow you to easily manage more than one value, and data typing those values to improve error reporting
- Logical structures such as conditionals for decision making and loops for simplifying repetitive tasks
- Functions that can isolate code into convenient blocks that will be executed only when instructed
- Ways to address Flash objects with ActionScript, including using absolute and relative paths, and the shortcut identifier `this`

CHAPTER 2

CORE LANGUAGE FUNDAMENTALS

It's true that ActionScript 3.0 is a complete rewrite of Flash's internal scripting language, and it's also true that ActionScript 3.0 doesn't share the same runtime code base as prior versions of ActionScript. But that's all behind the scenes. The truth is, all versions of ActionScript to date share quite a bit in common.

This is not hard to understand, since ActionScript was based on a scripting language standard (called ECMA-262) that grew from the success of JavaScript, and that ongoing versions of ActionScript are backward compatible to support legacy projects.

That is not to say that the language isn't growing. Certainly, each new version of ActionScript introduces a batch of newly supported features, as is true with the evolution of most programming languages. And, since the decision was made to write ActionScript 3.0 from the ground up, the opportunity presented itself to tidy up a few messy things that lingered from previous versions—namely, tightening up and requiring best practices that had been optional, and restructuring the event and display systems.

All of this progress, however, did not steamroll over the standard upon which ActionScript is based, and most of the language fundamentals remain intact. With the intention to focus on new ActionScript 3.0 options later on, we want to cover some of the more important fundamentals up-front. We do not intend to ignore these ideas throughout the rest of the book. However, because they are core fundamentals and are, therefore, used often, we hope to explain them in sufficient detail here and spend less time on them as we proceed.

If you're already comfortable with ActionScript and are reading this text for a head start learning version 3.0, you may want to skip, or at least skim, this chapter. It is by no means a comprehensive starter course. This book does not assume that you are well versed in any prior version of ActionScript, but its size and purpose requires that we assume a very basic understanding of general scripting concepts. If you haven't already, please look over the Preface for a good idea of who this book is for, as well as a few alternative references if you need more background information.

IN THIS CHAPTER

You can use this chapter, however, as a point of reference when an underlying programming concept needs further explanation. In these pages, we'll look at the following topics:

- **Miscellaneous Basics.** To identify some of the items and techniques used throughout this book that don't necessarily warrant a section for each, we'll start off with a few essentials.
- **Variables and Data Types.** Information must be stored in containers called variables if it is to be recalled for later use, and declaring which type of data will be stored in each variable can help Flash check for errors during development.
- **Conditionals.** Often, when a decision must be made in the course of a script's execution, a conditional is used to evaluate the outcome of a prescribed set of conditions. We'll look at the `if` and `switch` conditional forms.
- **Loops.** When you must execute an instruction multiple times, it is sometimes handy to do so within a loop structure. We'll look at the commonly used `for` loop structure, but also at alternatives to explicit loops, including frame and timer events.
- **Arrays.** While a basic variable can contain only a single value, it is frequently efficient, or even necessary, to store more than one value in a variable. Imagine a shopping list, with several items, written on a single piece of paper. The array is a data structure that allows you to store multiple values in a single variable.
- **Functions.** Functions are essential to just about any programming language, and allow you to execute code only when you are ready to do so, and reuse that code efficiently.
- **Custom Objects.** A custom object can be considered an advanced kind of variable that allows you to store lots of information, in a way that is easy and consistent to retrieve. Objects can also be very useful for simplifying the task of passing multiple optional values to a function.
- **`this`.** The `this` keyword is used as a shorthand reference, essentially meaning the object or scope in a script. This will become clearer in context, but understanding how the keyword works can save you much repetitive typing and reduce the need for more complex references in your scripts.
- **Absolute versus Relative Addresses.** ActionScript can reference addresses to its objects using absolute paths, such as starting from the root timeline and including every object between it and your destination, or relative paths, such as going up to a parent and down to a sibling, no matter where you are.

Again, this chapter is not meant to act as the only reference to bring you up to speed if you have absolutely no experience with ActionScript. It will likely serve the bulk of your needs but other basics—such as where scripts are stored in the Flash interface—have been omitted for space reasons.

As described in the Preface, for a starter book on the Flash interface, we recommend *Flash CS3 Professional, The Missing Manual*, published by O'Reilly, the publisher of this book. For a more complete ActionScript 3.0 resource, we heartily recommend the incomparable *Essential ActionScript 3.0* by Colin Moock, also published by O'Reilly. The latter is decidedly an intermediate to advanced reference but, at nearly three times the size of this volume, it is also substantially more comprehensive.

For the most part, this chapter, along with the context and supplemental explanations presented in subsequent chapters, should provide you with enough to understand the topics and to get the sample exercises working.

Miscellaneous Basics

Some basic topics probably don't require a section devoted to their discussion but should still be mentioned due to their use throughout the book. We'll include a few such examples here, just to get us started.

Execution order

In general, ActionScript executes in a top-to-bottom, left-to-right order—that is, each line executes one after another, working from left to right. Several things can change this order in subtle ways, but it's basically a reliable rule of thumb. For example, subroutines of one type or another can be called in the middle of a script, causing the execution order of the original script to pause while the remote routine is executed. When the subroutine has completed, the execution of the original script continues where it left off. These steps will be explained in context, in all scripts in this book.

Use of the semicolon(;)

The official use of the semicolon in ActionScript is to execute more than one instruction on a single line. This is rare in the average script, and we will look at this technique when discussing loops. However, the semicolon is also used to indicate the end of a line. This is not required, but it is recommended for clarity and to ease any possible transition into learning other languages in which the semicolon at the end of a line is required.

Evaluating an expression

It's helpful to note that you are usually not solving an equation when you see an expression with like values on the left and right of an equal sign. For example if you see something like `x = x + 1`, it is unlikely that you will be solving for the value of `x`. Instead, this line is assigning a new value to `x` by adding `1` to its previous value.

Use of the `trace` command

As a means of getting quick feedback in an example, or as a testing and debugging technique when writing scripts, it is very helpful to use the `trace` command. This instruction places any relevant text into the Output panel of the Flash interface. As such, this is an option that is available only at author-time, and has no use at runtime.

Variables and Data Types

Variables are best described as containers into which you place information for later recall. Imagine if you were unable to store any information for later use. You would not be able to compare values against previously described information (such as user names or passwords), your scripts would suffer performance lags due to repeated unnecessary calculations, and you wouldn't be able to carry any prior experiences through to the next possible implementation of a task. In general, you wouldn't be able to do anything that required data that your application had to "remember."

Variables make all this and more possible, and are relatively straightforward. In basic terms, you need only create a variable with a unique name, so it can be referenced separately from other variables and the ActionScript language itself, and then populate it with a value. A simple example is remembering the number 1 with the following:

```
myVariable = 1;
```

There are just a few rules and best practices to consider when naming variables. They must be one word, can only include alphanumeric characters along with the dollar sign ($) or underscore (_), should not start with a number, and should not already be a keyword or reserved word in ActionScript.

To help ensure that you are using variables appropriately, ActionScript will monitor them and warn you if you are trying to perform an illegal operation on them, or otherwise use them incorrectly. For example, if you try to perform a mathematical operation on a passage of text, Flash will issue a warning so you can correct the problem.

To do this, Flash must be told what you intend to store in each variable. This is accomplished by declaring the variable by preceding its first use with the `var` keyword, and citing the type of data to be stored therein by following the name of the variable with a colon (:) and data type. For instance, the previous example of remembering the number 1 should be written this way:

```
var myVariable:Number = 1;
```

There are several native data types including, but not limited to, those listed in Table 2-1:

Table 2-1. Variable types

Data type	Example	Description
`Number`	`4.5`	Any number, including floating point values (decimals)
`int`	`-5`	Any integer or whole number
`uint`	`1`	Unsigned integer or any non-negative whole number
`String`	`"hello"`	Text or a string of characters
`Boolean`	`true`	True or false
`Array`	`[2, 9, 17]`	More than one value in a single variable
`Object`	`myObject`	The basic structure of every ActionScript entity, but also a custom form that can be used to store multiple values as an alternative to `Array`.

There are also many dozens of additional data types that describe which class was used to populate the variable. (As discussed in Chapter 1, think of classes as external scripts that typically return information to your script and work as members of a larger team to create your application.) For example, the following line of code uses the `MovieClip` class (built into Flash) to create a movie clip at runtime:

```
var myMC:MovieClip = new MovieClip();
```

It is impractical to list every possible data type here, but we will reference data types frequently throughout the book, and it will soon become second nature to use them.

In previous versions of ActionScript, declaring and typing variables was optional. However, in ActionScript 3.0, this practice is required. This may seem cumbersome but, before long, this will become second nature, and you will come to appreciate the instant error checking and feedback this feature provides.

As we get further into the book, you'll learn that variables can apply to an entire scope (the realm in which the variable lives, such as Flash's main timeline, or a particular class) or be local to specific code structures. We will discuss this in context in the code examples.

Conditionals

You will often have the need to make a decision in your script, choosing to do one thing under one circumstance, and another thing under a different circumstance. These situations are usually addressed by *conditionals*. Put simply, a test is created, asking whether a condition is met. If the condition is met, the test evaluates to `true`, and specific code is executed accordingly. If the condition is not met, either no further action is taken or an alternate set of code is executed.

NOTE

Throughout this book, the code examples are syntax-colored in the same way that the Flash interface colors scripts. This helps identify colored items as part of the ActionScript lexicon (such as keywords and identifiers) and makes it a bit easier to see comments (descriptive text passages that are not executed) and strings.

if

The most common form of the conditional is the `if` statement. The statement's basic structure is the `if` keyword, followed by parentheses in which the conditional test resides, and braces in which the code resides that is executed when the statement evaluates to `true`. The first three lines in the following example establish a set of facts. The `if` statement evaluates the given facts. (This initial set of facts will be used for this and subsequent examples in this section.)

```
var a:Number = 1;
var b:String = "hello";
var c:Boolean = false;

if (a == 1) {
    trace("option a");
}
```

NOTE

The test in this example uses a double equal sign. This is a comparison operator that asks, "Is this equal to?" This distinction is very important because the accidental use of a single equal sign will cause unexpected results. A single equal sign is an assignment operator and assigns the value in the right side of the equation to the object in the left side of the equation. Because this assignment occurs, the test will always evaluate to true.

To evaluate the truth of the test inside the parentheses, conditionals often make use of *comparison* and *logical operators*. A comparison operator compares two values, such as equals (==), less than (<), and greater than or equal to (>=), to name a few.

A logical operator evaluates the logic of an expression. Included in this category are AND (`&&`), OR (`||`), and NOT (`!`). These allow you to ask if "this *and* that" are true, or if "this *or* that" are true, or if "this" is *not* true.

For example, the following would be false, because *both* conditions are not true. As a result, nothing would appear in the Output panel.

```
if (a == 1 && b == "goodbye") {
    trace("options a and b");
}
```

In this example, the test would evaluate to true, because *one* of the two conditions (the first) is true. As a result, "option a or b" would be traced.

```
if (a == 1 || b == "goodbye") {
    trace("option a or b");
}
```

Finally, the following would also evaluate to true, because the NOT operator correctly determines that `c` is not true. (Remember, that every `if` statement, at its core, is testing for truth.)

```
if (!c) {
    trace("not option c");
}
```

The NOT operator is also used as part of a comparison operator. When combined with a single equal sign, the pair means, "not equal to." Therefore, the following will fail because `a` does equal 1, and nothing will be traced.

```
if (a != 1) {
    trace("a does not equal 1");
}
```

Additional power can be added to the **if** statement by adding an unconditional alternative (true no matter what)—that is, an alternative set of code is executed no matter what the value being tested is, simply because the test did not pass. With the following new code added to the previous example, the last trace will occur.

```
if (a != 1) {
    trace("a does not equal 1");
} else {
    trace("a does equal 1");
}
```

Finally, the statement can be even more robust by adding a conditional alternative (or an additional test), to the structure. In this example, the second trace will occur.

```
if (a == 2) {
    trace("a does not equal 1");
} else if (a == 1) {
    trace("a does equal 1");
}
```

The **if** statement requires one **if**, only one optional **else** can be used, and any number of optional additional **else if** tests can be used. In all cases, however, only one result can come from the structure. Consider the following example, in which all three results could potentially execute—the first two because they are true, and the last because it is an unconditional alternative.

```
if (a == 1) {
    trace("option a");
} else if (b == "hello") {
    trace("option b");
} else {
    trace("option other");
}
```

In this case, only "option a" would appear in the Output panel because the first truth would exit the **if** structure. If you needed more than one execution to occur, you would need to use two or more conditionals. The following structure, for example, executes the first trace in each **if**, by design.

```
if (a == 1) {
    trace("option a");
}
if (b == "hello") {
    trace("option b");
} else {
    trace("option other");
}
```

switch

An **if** statement can be as simple or as complex as needed. However, long **if** structures can be difficult to read and can sometimes better be expressed using the **switch** statement. In addition, the latter statement has a unique

feature that lets you control which if *any* instructions are executed—even when a test evaluates to false.

Imagine an **if** statement asking if a variable is 1, **else if** it's 2, **else if** it's 3, **else if** it's 4, and so on. A test like that can become difficult to read quickly. An alternate structure appears as follows:

```
switch (a) {
    case 1 :
    trace("one");
    break;
    case 2 :
    trace("two");
    break;
    case 3 :
    trace("three");
    break;
    default :
    trace("other");
    break;
}
```

In this case, "one" would appear in the Output panel. The **switch** line contains the object or expression you want to test. Each case line offers a possible value. Following the colon are the instructions to execute upon a successful test, and each **break** line prevents any following instructions from executing. When not used, the next instructions in line will execute, even if that test is false.

For example, the following will place both "one" and "two" in the Output panel, even though **a** does not equal 2.

```
switch (a) {
    case 1 :
    trace("one");
    case 2 :
    trace("two");
    break;
}
```

This **break** feature does not exist with the **if** statement and, if used with care, makes **switch** an efficient alternative to a more complex series of multiple **if** statements. **Switch** statements must have one **switch** and one **case**, an optional unconditional alternative in the form of **default**, and an optional **break** for each **case** and **default**. The last **break** is not needed, but may be preferred for consistency.

Loops

It is quite common to execute many repetitive instructions in your scripts. However, including them line by line, one copy after another, is inefficient and difficult to edit and maintain. Wrapping repetitive tasks in an efficient structure is the role of *loops*. A programming loop is probably just what you

think it is: Use it to go through the structure and then loop back to the start and do it again. There are a few kinds of loops, and the type you choose to use can help determine how many times your instructions are executed.

for Loop

The first type of loop structure we'll look at is the **for** loop. This loop executes its contents a finite number of times. For example, you may wish to create a grid of 25 movie clips or check to see which of 5 radio buttons has been selected. In our first example, we want to trace content to the Output panel three times.

To loop through a process effectively, you must first start with an initial value, such as 0, so you know you have not yet traced anything to the Output panel. The next step is to test to see whether you have exceeded your limit. The first time through, 0 does not exceed the limit of three times. The next step is to trace the content once, and the final step is to increment your initial value, registering that you've traced the desired content once. The process then starts over until, ultimately, you will exceed the limit of the loop. The syntax for a basic **for** loop is as follows:

```
for (var i:Number = 0; i < 3; i++) {
    trace("hello");
}
```

The first thing you may notice is the declaration and typing of the counter, **i**. This is a common technique because the **i** variable is often used only for counting and, therefore, is created on the spot and not used again. If you have already declared and typed the counter previously, that step can be omitted here. Next is the loop test. In this case, the counter variable must have a value that is less than 3. Finally, the double-plus sign (**++**) is equivalent to i = i + 1, or add 1 to the current value of **i**. The result is three occurrences of the word "hello" in the Output panel.

It is also possible to count down by reversing the values in steps 1 and 2, and then decrementing the counter:

```
for (var i:Number = 3; i > 0; i--) {
    trace("hello");
}
```

In other words, as long as the value of **i** is greater than 0, execute the loop, and subtract one from the counter each time. This is less common, and works in this case because the loop only traces a string. However, if you need to use the actual value of **i** inside the loop, that need may dictate whether you count up or down. For example, if you created 10 movie clips and called them **mc0**, **mc1**, **mc2**, and so on, it may be clearer to count up.

NOTE

Note in each example loop the "official" use of the semicolon to execute more than one step in a single line.

WARNING

*Use **while** loops with caution until you are comfortable with them. It's very easy to accidentally write an infinite loop with no exit, which will cause your code to stop in its tracks. Do not try this code yourself, but here is a significantly simplified example of an infinite loop:*

```
var flag:Boolean = true;
while (flag) {
    trace ("infinite loop");
}
```

As you can see from this example, the flag variable remains true and, therefore, the loop can never fail.

while Loop

The other kind of loop that you are likely to use is a **while** loop. Instead of executing its contents a finite number of times, it executes as long as something remains true. As an example, let's look at a very simple case of choosing a random number. Using the **Math** class's **random()** method, ActionScript chooses a random number between 0 and 1. So, let's say you wanted to choose a random number greater than or equal to .5. With essentially a 50-percent chance of choosing a desired number each time, you may end up with the wrong choice several times in a row. To be sure you get a qualifying number, you can add this to your script:

```
var num:Number = 0;
while (num < .5) {
    num = Math.random();
}
```

Starting with a default value of 0, **num** will be less than .5 the first time into the loop. A random number is then put into the **num** variable and, if it's less than .5, the loop will execute again. This will go on until a random number that is greater than .5 is chosen, thus exiting the loop.

A Loop Caveat

It's very important to understand that, although compact and convenient, loop structures are not always the best method to use to achieve an outcome. This is because loops are very processor intensive. Once a loop begins its process, nothing else will execute until the loop has been exited. For this reason, it may be wise to avoid **for** and **while** loops when interim visual updates are required.

In other words, when a loop serves as an initialization for a process that is updated only once upon its completion, such as creating the aforementioned grid of 25 movie clips, you are less likely to have a problem. The script enters the loop, 25 clips are created, the loop is completed, a frame update can then occur, and you see all 25 clips.

However, if you want each of the 25 clips to appear, one by one, those interim visual updates of the playhead cannot occur while the processor is consumed by the loop. In this situation, a loop that is achieved by other means—methods that do not interfere with the normal playhead updates—is desirable. Two such loops, frame and timer loops, are commonly used for this purpose. A frame loop is simply a repeating frame event, executing an instruction each time the playhead is updated. A timer loop is similar, but is not tied to the frame tempo. Instead, a timer event is triggered by an independent timer at a set frequency.

In both cases, the events occur in concert with any other events in the ordinary functioning of the file, so visual updates, as one example, can continue to occur. Both frame and timer loops will be explained, complete with examples, in the next chapter.

Arrays

Basic variables can contain only one value. If you set a variable to 1 and then set that same variable to 2 in the following line, the value would be reassigned, and the value of the variable would be 2.

However, there are times when you need one variable to contain more than one value. Think of a hypothetical set of groceries, including 50 items. The standard variable approach to this problem would be to define 50 variables and populate each with a grocery item. That is the equivalent of 50 pieces of paper, each with one grocery item written on its face. This is unwieldy and can only be created at author time—at which point the process is fixed—and you'd have to recall and manage all variable names every time you wanted to access the grocery items.

An array equivalent, however, is very much like how we handle this in real life. A list of 50 grocery items is written on one piece of paper. You can add to the list while at the store, cross each item off once it is acquired, and you only have to manage one piece of paper.

Creating an array is quite easy. You can prepopulate an array by setting a variable (typed as an **Array**) to a comma-separated list of items, surrounded by brackets. You can also create an empty array by using the **Array** class. Both techniques are illustrated here:

```
var myArray:Array = [1, 2, 3]
var yourArray:Array = new Array();
```

In both cases, you can add to, or remove from, the array at runtime. For example, you can add a value to an array using the **push()** method, which pushes the value into the array at the end. In short, a *method* is an action performed by an object—in this case adding something to the array—and will be discussed in detail in the next chapter. You can remove an item from the end of an array using the **pop()** method.

```
var myArray:Array = new Array();
myArray.push(1);
trace(myArray)
// 1 appears in the Output panel
myArray.push(2);
// the array now has two items: 1, 2
trace(myArray.pop());
// the pop() method removes the last item, displaying its value of 2
trace(myArray)
// the lone remaining item in the array, 1, is displayed
```

There are a dozen or so other array methods, allowing you to add to or remove from the front of an array, sort its contents, check for the position of a found item within the array, compare each value against a control value, and more.

NOTE

Both methods are added to the end of the **myArray** *variable with a dot separating the two. This is the syntax used to navigate the Flash document object model, and is sometimes referred to as dot syntax. Essentially, this system strings together a series of items, from biggest to smallest, and including only items relevant to the task at hand. In this case, the largest relevant item is the array itself and, below that is each method. Considering another example, where you may wish to check the width of a movie clip that is inside another movie clip, the biggest item will be the parent, or container movie clip, then comes the nested movie clip, and then comes its width:*

```
mc1.mc2.width;
```

This dot syntax will be used in virtually every example for the rest of the book, and it will soon become quite easy to understand just what is referenced by each object along the way.

You can also add to or retrieve values from locations within the array, by using brackets and including the index, or position, of the array you need. To do so, you must understand that ActionScript uses zero-based arrays, which means that the first value is at position 0, the second is at position 1, the next at position 2, and so on. As an example, to retrieve the existing fifth value from an array, you must request the item at position 4.

```
var myArray:Array = ["a", "b", "c", "d", "e"]
trace(myArray[4])
//"e" appears in the Output panel
```

There are other kinds of arrays, such as multidimensional arrays (arrays within arrays that can resemble database structures) and associative arrays (which store not only linear values, but also a linear *pair* of items—the value and a property name to describe that value), for example. However, due to space constraints, we've focused on the most common array type: the linear array. Any other uses of array structures will be highlighted in future chapters.

Functions

Functions are an indispensable part of programming in that they wrap code into blocks that can be executed only when needed. They also allow code blocks to be reused and edited efficiently, without having to copy, paste, and edit repeatedly. Without functions, all code would be executed in a linear progression from start to finish, and edits would require changes to every single occurrence of any repeated code.

Creating a basic function requires little more than surrounding the code you wish to trigger at will with a simple syntax that allows you to give the block a name. Triggering that function later requires only that you call the function by name. The following syntax shows a function that traces a string to the Output panel. The function is defined and then, to illustrate the process, immediately called. (In a real-world scenario, the function is usually called at some other time or from some other place, such as when the user clicks a button with the mouse.) The output is depicted in the comment that follows the function call.

```
function showMsg(){
    trace("hello");
}
showMsg();
//hello
```

If reusing code and executing code only when needed were the only advantage of functions, you'd already have a useful enhancement to linear execution of ActionScript, because it would allow you to group your code into subroutines that could be triggered at any time and in any order. However, you can do much more with functions to gain even greater power.

For example, assume you need to vary the purpose of the previous function slightly. Let's say you need to trace ten different messages. To do that without any new features, you'd have to create ten functions and vary the string that is sent to the Output panel in each function.

However, this can be more easily accomplished with the use of *arguments*, or very local variables that have life only within their own functions. By adding an argument to the function declaration, in this case the string argument **msg**, you can pass a value into that argument when you call the function. By using the argument in the body of the function, it takes on whatever value was sent in. In this example, the function no longer traces "hello" every time it is called. Instead, it traces whatever text is sent into its argument when the function is called. When using arguments, it is necessary to type the data coming in so Flash knows how to react and can issue any warnings needed to notify you of errors.

```
function showMsg(msg:String) {
    trace(msg);
}
showMsg("goodbye");
//goodbye
```

It is also possible to return a value from a function, increasing its usefulness. Having the ability to return a value to the script from which it was called means a function can vary its input *and* its output. Included below are examples to convert temperature values from Celsius to Fahrenheit and Fahrenheit to Celsius. In both cases, a value is sent into the function and a resulting calculation is returned to the script. The first example immediately traces the result, while the second example stores the value in a variable. This mimics real-life usage in that you can immediately act upon the returned value or store and process it at a later time.

```
function celToFar(cel:Number):Number {
    return (9/5)*cel + 32;
}
trace(celToFar(20));
//68

function farToCel(far:Number):Number {
    return (5/9)*(far - 32);
}
var celDeg:Number = farToCel(68);
trace(celDeg);
//20
```

Note that, when returning a value from a function, you should also declare the data type of the return value. This is achieved the same way as when applying other data types—with a colon followed by the type specific to that function—and this form is placed between the argument close parenthesis and the opening function brace. Once you get used to this practice, it is best to specify **void** as a return type when your function does not return a value.

Custom Objects

After just a short while working with ActionScript, you will realize that you are immersed neck-deep in objects. Most discrete entities in ActionScript are descendents of the Object class and tend to behave in a consistent reliable manner. Central to this behavior is the ability for an object to have properties (which are essentially descriptive elements that contribute to the object's general characteristics, like width, location, rotation, and so on), methods (which are actions the object can perform), and even events (custom events that, like a mouse click or a key press, can trigger other processes in the course of working with a script).

You can also create custom objects and define your own properties, methods, and events. To demonstrate this, we'll create a custom object called plane, and give it properties for pitch, roll, and yaw. These properties are terms that describe rotation in 3D space. If you think of yourself seated in a plane, pitch is the angle of rotation that would cause the nose of the plane to go down or up. Roll is the angle of rotation that would cause the plane to spin along the length of the plane, keeping the nose facing forward as you spiral through flight. Finally, yaw is the angle of rotation that comes up perpendicularly through your seat on the plane, causing the plane to spin in a flat spin where the nose would no longer remain facing forward.

None of these terms—plane, pitch, roll, or yaw—are part of the ActionScript library. However, by creating a custom object, we will temporarily make them available to our scripts as if they were always there. The first step in this process is to create the object. Once created, we can add and populate properties:

```
var plane:Object = new Object();
plane.pitch = 0;
plane.roll = 5;
plane.yaw = 5;
```

These values would send the plane in a slow right-hand turn. They can be called up at any time, by querying the properties the same way they were created.

```
trace(plane.pitch);
//0
```

Creating a custom object to contain properties is a highly effective way of sending multiple optional parameters into a function. ActionScript 3.0 does not like having a variable number of arguments or values for those arguments. If you specify five arguments, it expects five parameters and will balk if you choose to omit any. If you plan your code ahead and plan to allow a series of optional parameters, it is easy to transmit an unknown number of parameter values through a fixed single argument that contains an object. You can then parse the values from this object inside the function, initializing the starting value of any specific properties that were omitted. Here is an example, using the previously created **plane** object:

```
function showPlaneStatus(obj:Object):void {
    trace(obj.pitch);
    trace(obj.roll);
    trace(obj.yaw);
};
showPlaneStatus(plane);
//0
//5
//5
```

this

Although a bit nebulous for some just starting with ActionScript, **this** can be your friend. It is essentially shorthand for "whichever object or scope you're working with now." *Scope* is the realm or space within which an object lives. For example, think of a movie clip inside Flash's main timeline. Each of these objects has a unique scope, so a variable or function defined inside the movie clip will not exist in the main timeline, and vice versa.

It is easiest to understand the usage of **this** in context, but here are a couple of examples to get you started. If you wanted to refer to the width of a nested movie clip called **mc** from within the main timeline, you might say:

```
this.mc.width;
```

If you wanted to refer to the main timeline from the nested movie clip, you might write:

```
this.parent.mc.width;
```

In both cases, **this** is a reference point from which you start your path. It is fairly common to drop the keyword when going down from the current scope (as in the first example), but it is required when going up to a higher scope (as in the second example). This is because Flash must understand what the parent is actually a parent *of* in order to start traversing through the hierarchy. Imagine a family reunion in which several extended family members, including cousins and multiple generations, are present, and you are looking for your mother, father, or grandparent. If you just said "parent," any number of parents might answer. If you, instead, said "my parent" or "my mother's parent," that would be specific enough to get you headed in the right direction.

Absolute versus Relative Addresses

Much like a computer operating system's directory, or the file structure of a web site, ActionScript refers to the address of its objects in a hierarchical fashion. You can reference an object address using an absolute or relative path. Absolute paths can be easy because you most likely know the exact path to any object starting from the main timeline. However, they are quite rigid and will break if you change the nested relationship of any of the referenced objects. Relative paths can be a bit harder to call to mind at any given

moment, but they are quite flexible. Working from a movie clip and going up one level to its parent and down one level to a sibling will work from anywhere—be that in the root timeline, another movie clip, or nested even deeper—because the various stages aren't named.

Table 2-2 and Table 2-3 draw parallels to the operating system and web site analogies:

Table 2-2. Absolute (from main timeline to nested movie clip)

ActionScript	Windows OS	Mac OS	Web Site
`root.mc1.mc2`	*c:\folder1\folder2*	*Macintosh/folder1/folder2*	`http://www.domain.com/dir/dir`

Table 2-3. Relative (from a third movie clip, up to the root, and down to the child of a sibling)

ActionScript	Windows OS	Mac OS	Web Site
`this.parent.mc1.mc2`	..\folder1\folder2	../folder1/folder2	`../dir/dir`

What's Next?

Ideally, we've provided just enough background (or review) of key ActionScript fundamentals to now focus in on topical syntax. Although we won't entirely ignore basic elements within the scripts of future chapters, we will spend more time describing the collective goal of a script, and highlighting new issues introduced or updated by ActionScript 3.0.

Next, we start off the ActionScript 3.0-specific material with a look at the three essential building blocks of most ActionScript objects: properties, methods, and events—the latter being one of the most significantly changed elements of ActionScript, with the introduction of version 3.0.

In the next chapter, we'll discuss:

- The descriptive properties (such as width, height, location, alpha (opacity), rotation, and more) of each object that define its major characteristics
- The actions you may exert on objects, or that objects may take on other objects, in the form of methods
- The events issued by the user, or aspects of your program or environment, and, perhaps more directly, the *reactions* to those events

GRAPHICS AND INTERACTION

PART II

Part II represents the largest section of the book, spanning Chapter 3 through Chapter 9. This part covers many significant features that distinguish ActionScript 3.0 from prior versions. It focuses on graphics and interactions and includes the new event model and display list.

Chapter 3 is a discussion of properties, events, and methods—the items responsible for manipulating just about anything in Flash. Of particular importance is a section that describes a novel approach to handling events in ActionScript. Chapter 4 goes on to explain the display list, a great new way to display visual assets in Flash. Chapter 5 discusses timeline control, including various navigation techniques.

Chapter 6 marks an important transition in the book, as the remaining chapters in this part begin to focus more on object-oriented programming. Chapter 7 takes a look at various ways to animate graphics with ActionScript. Chapter 8 and 9 round out the presentation of graphics and interactivity with tutorials covering drawing with vectors and pixels. Included are demonstrations for creating vectors with ActionScript and manipulating a variety of bitmap properties in your projects.

IN THIS PART

CHAPTER 3

PROPERTIES, METHODS, AND EVENTS

In addition to the core language fundamentals reviewed in the previous chapter, you will find that the majority of your scripts are written using properties, methods, and events. These are the basic building blocks of most scripted tasks and allow you to get and set characteristics of, issue instructions to, and react to input from, many Flash elements.

- **Properties.** Properties are somewhat akin to adjectives in that they describe the object being modified or queried. For example, you can check or set the width of a button. Most properties are read-write, in that you can both get and set their values. Some properties, however, are read-only, which means you can ask for, but not change, their values.
- **Methods.** Methods are a bit like verbs. They are used to tell objects to do something, such as play and stop. In some cases, methods can be used to simplify the setting of properties. You might use a method called `setSize()`, for example, to simultaneously set the width and height of something. Other methods are more unique, such as `navigateToURL()`, which instructs a browser to display a web page.
- **Events.** Events are the catalysts that trigger the actions you write, setting properties and calling methods. For instance, a user might click the mouse button, which would then result in a mouse event. That event then causes a function to execute, performing the desired actions. *Event handlers* are the ActionScript middlemen that trap the events and actually call the functions. ActionScript 3.0 has unified event handling into a consistent system of what are called *event listeners*, which are set up to listen for the occurrence of a specific event and react accordingly.

In this chapter, you will build a utility that will demonstrate each of these ActionScript structures. By creating mouse and keyboard events, you will manipulate several common properties, as well as execute a few methods. The vast majority of ActionScript entities have properties, methods, and events. For clarity, we will focus primarily on the movie clip. Using the movie clip to centralize our discussion will make it easier for you to consult the Flash help system, online resources, and supplemental texts for additional information, as you look for other attributes to manipulate.

IN THIS CHAPTER

Inherited Attributes

One of the most important things to understand when consulting attributes is that ActionScript entities often share attributes in common with other entities. One reason for this is that they may be related in some way, such as being descendents from a common parent. In this case, the child inherits attributes from its parent. We introduced this concept a bit in Chapter 1 when we talked about classes. Consider the idea that a daughter, by virtue of being a different sex than her father, has several characteristics, or properties, that are distinct from her father. However, they also may share several characteristics in common, such as eye and hair color.

We will look at the sharing of attributes in greater depth throughout this book but, for now, all you need to know is that ActionScript reference materials are often organized by classes, and it would be redundant and cumbersome to list the same properties for every related class. Considering the movie clip, for example, every Flash element that can be displayed on stage—the movie clip among them—can have an x and a y coordinate, or location, on the stage. Listing these properties for every such item would eat up a lot of space and make the resource a bit harder to wade through.

To simplify things, the `x` and `y` properties are typically listed as inherited properties, as is true in the Flash help system. To view inherited properties, for example, in the Flash help system, just click the Show Inherited Public Properties link found immediately under the Public Properties header.

Properties

If you think of properties as ways of describing an object, they become second nature. Asking where a movie clip is, for example, or setting its width are both descriptive steps that both use properties.

In Chapter 2, we briefly discussed the object model and dot syntax that brings order and structure to ActionScript as well as many other scripting and programming languages. Referencing a property begins with an instance—let's call our square movie clip "box"—because you must decide which element you wish to query or change. If we consider a test file with only one movie clip in it, instantiated as "box," all that remains is referencing the property and either getting or setting its value.

To begin, we'll show you the syntax for making five changes to movie clip properties in the following table. Then, when we demonstrate how to handle events in the next section, we'll change these properties interactively. The following examples assume a movie clip of a square is on the stage, and has an instance name of "box." Figure 3-1 demonstrates the visual change made by each property. The light colored square is the original state when the movie

clip is moved. (The **alpha** property shows only the final state.) The dashed stroke for the visible property is only to show that the box is not visible.

Table 3-1 represents six movie clip properties with sample syntax and notes regarding each property's unit of measure and possible sample range of values.

Table 3-1. Movie clip properties

Description	Property	Syntax for Setting Value	Units and/or Range
Location	x, y	`box.x = 100;` `box.y = 100;`	pixels
Scale (1)	scaleX, scaleY	`box.scaleX = .5;` `box.scaleY = .5;`	percent / 0-1
Scale (2)	width, height	`box.width = 72;` `box.height = 72;`	pixels
Rotation	rotation	`box.rotation = 45;`	degrees / 0-360
Transparency	alpha	`box.alpha = .5;`	percent / 0-1
Visibility	visible	`box.visible = false;`	Boolean

If you have experience with prior versions of ActionScript, you may notice a few changes in the property syntax. First, the properties do not begin with an underscore. This is a beneficial consistency introduced with ActionScript 3.0. Rather than varying property syntax, some with and some without leading underscores, no properties begin with the underscore character.

Second, some value ranges that used to be 0–100 are now 0–1. Examples include **scaleX**, **scaleY**, and **alpha**. Instead of using 50 to set a 50% value, specify .5.

Finally, the first scaling method uses properties **scaleX** and **scaleY**, rather than **_xscale** and **_yscale**, which are their AS1/AS2 equivalents. Typically, AS3 properties will cite the x and y version of a property as a suffix, to make referencing the property easier.

Table 3-1 shows syntax for setting a property. Querying the value of a property, also known as getting the property, is just as easy. For example, if you wanted to trace the box's alpha value, or store it in a variable, you could write either of the following:

```
trace(box.alpha);
var bAlpha:Number = box.alpha;
```

You can also use compound assignment operators to easily update the values. The following code will add 20 degrees to the current value of the box's rotation.

```
box.rotation += 20;
```

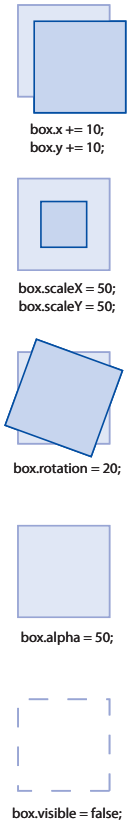

Figure 3-1. Changes to five movie clip properties

Events

Events make the Flash world go 'round. They are responsible for setting your scripts in motion, causing them to execute. A button can be triggered by a mouse event, text fields react to keyboard events—even calling your own custom functions is a means of issuing a custom event.

Events come in many varieties. In addition to the obvious events like mouse and keyboard input, most ActionScript classes have their own events. For example, events are fired when watching a video, working with text, and resizing the stage. To take advantage of these events to drive your application, you need to be able to detect their occurrences.

In previous versions of ActionScript, there were a variety of ways to trap events. You could apply a script directly to a button, for example, and use the **on(Release)** approach. As the language matured, you could create event handlers and apply them remotely using instance names, using **myButton.onRelease** for example. Finally, you could use event listeners, primarily with components or custom objects.

In ActionScript 3.0, trapping events is simplified by relying on one approach for all event handling, which is to use event listeners regardless of the type of event or how it is used. The **EventDispatcher** class that "oversees" event listeners is not new, but it has been improved and is now responsible for handling the majority of events in AS3.

The **EventDispatcher** class allows you to listen for the occurrence of events by putting event listeners into service, clean up your code by removing unneeded listeners from service, and manually dispatching events when you need an event to occur at a specific time. You can also check to see whether an object has a listener already set up for a specific event, which we'll look at later when we talk about event propagation.

Using Event Listeners

The concept of event listeners is pretty simple. Imagine that you are in a lecture hall that holds 100 people. Only one person in the audience has been given instructions about how to respond when the lecturer asks a specific question. In this case, one person has been told to listen for a specific event, and to act on the instructions provided when this event occurs.

Now imagine that many more responses need to be planned. For example, when the lecturer takes the stage, the lights must be dimmed. When the lecturer clicks a hand-held beeping device, an audio/visual technician must advance to the next video in the presentation. When each video ends, the lecturer must react by introducing the next exhibit in the lecture. Finally, when an audience member raises a hand, an usher must bring a microphone to assist the audience member in asking his or her question.

These are all reactions to specific events that are occurring throughout the lecture. Some are planned and directed to a specific recipient—such as the beeping that triggers the technician to advance to the next video in the series. Others are unplanned, such as when, or even if, an audience member has a question. Yet each appropriate party in the mix has been told which event to listen for and how to react when that event occurs.

Creating an event listener, in its most basic form, is fairly straightforward. The nuances that make the process anything more than simple add power to the system and can be used to your advantage. The first main step is to identify the host of the listener—that is, who should be told to listen for a specific event. One easy-to-understand example is that a button should be told to listen for mouse events that might trigger its scripted behavior.

Once you have identified an element that should listen for an event, the next major step is choosing an event appropriate for that element. For example, it makes sense for a button to listen for a mouse event, but it makes less sense for the same button to listen for the end of a video or the resizing of the stage. It would be more appropriate for the video player to listen for the end of the video, and the stage to listen for any resize event. Each respective element could then act, or instruct others to act, when that event occurs—which is the third main step in setting up a listener.

To identify the instructions that must be executed when an event occurs, you simply need to write a function and tell the event listener to call that function when the event fires. That function uses an argument to receive information about the event that called it, allowing the function to use key bits of data during its execution.

To tie it all together, the **addEventListener()** method is used to identify the event, and assign the function to be executed to the object that is supposed to be doing the listening. Let's go back to the button example, in which the button should listen for a **mouse up** event. Let's say the button is called *rotate_right_btn*, and the function it should execute is **onRotateRight()**. The code would look something like this:

```
1  rotate_right_btn.addEventListener(MouseEvent.MOUSE_UP,
   onRotateRight);
2  function onRotateRight(evt:MouseEvent):void {
3      box.rotation += 20;
4  }
```

In line 1, you start with the button instance name and then add the **addEventListener()** method. The method requires two mandatory parameters. The first is the event for which you want to listen. Each event you are trying to trap, be it a built-in event or a custom event of your own making, originates in a class that defines that event. Built-in events are typically found in classes dedicated specifically to events, and the event itself is usually defined as a constant in that class. For example, the **MouseEvent** class contains constants that refer to mouse events like **mouse up** and **mouse down**. This

example uses the **MOUSE_UP** constant to reference the **mouse up** event. Other examples include the **ENTER_FRAME** constant in the **Event** class, for simulating playhead updates, and the **KEY_UP** event in the **KeyboardEvent** class, for trapping user keyboard input. We'll look at both of these events later on in this chapter.

The second parameter is the function that should be called when the event is received. In this example, a reference to the **onRotateRight()** function, defined in lines 2 through 4, is specified. You will probably be familiar with the structure of the function from the discussion about functions in Chapter 2. To review, the function contents are defined by the braces. In this case, line 3 adds 20 degrees to the current rotation value of the movie clip "box." Also explained in Chapter 2, the *void* that follows the function name and parentheses indicates that no value is returned by the function.

What hasn't been fully explained is the argument of the function that receives that event. Unlike custom functions, the argument in listener functions is required. In the following code example, it is arbitrarily named **evt** and receives information about the element that triggered the event. If helpful, you can parse information from this argument for use in the function, which you'll see below. The argument must be typed to the expected data. This will help you find errors if an incorrect event type is received. In this case, because we're listening for a `MouseEvent`, that is the data type used for the argument.

To illustrate this, let's look at another mouse event example. This time, however, we'll view multiple events, and parse information from the argument to show some of the benefits of this structure.

```
1  myMovieClip.addEventListener(MouseEvent.MOUSE_DOWN, onStartDrag);
2  myMovieClip.addEventListener(MouseEvent.MOUSE_UP, onStopDrag);
3  function onStartDrag(evt:MouseEvent):void {
4      evt.target.startDrag();
5  }
6  function onStopDrag(evt:MouseEvent):void {
7      evt.target.stopDrag();
8  }
```

In this example, two event listeners are assigned to a movie clip. One listens for a **mouse down** event, another listens for **mouse up**. They each invoke different functions. In both functions, however, the **target** property of the event, which is sought from the function argument, is used to identify which element received the mouse event. This allows the function in line 3 to start dragging the movie clip that was clicked, and also allows the function in line 6 to stop dragging the movie clip that was clicked, both without specifying the movie clip by its instance name. This generic approach is very useful because it makes the function much more flexible. The function can act upon any appropriate item that is clicked and passed into its argument. In other words, the same function could start and stop dragging any movie clip to which the same listener was added.

In the accompanying source files, the *start_stop_drag.fla* file shows this by adding the following lines to the previous example:

```
9   myMovieClip2.addEventListener(MouseEvent.MOUSE_DOWN, onStartDrag);
10  myMovieClip2.addEventListener(MouseEvent.MOUSE_UP, onStopDrag);
```

Simply by adding another movie clip to the exercise, and specifying the same listeners, you can drag and drop each movie clip.

Using Mouse Events to Control Properties

Now we can combine the syntax we've covered in the "Properties" and "Events" sections, to set up interactive control over properties. In the *chapter03* directory of the accompanying source code for this book, you'll find a file called *props_events.fla*. It contains nothing more than the example movie clip "box," and two buttons in the library that will be used repeatedly to change the five properties discussed earlier. The movie clip contains numbers to show which of its frames is visible at any time, and the instance names of each button will reflect its purpose. Included are *move_up_btn*, *scale_down_btn*, *rotate_right_btn*, *fade_up_btn*, and *toggle_visibility_btn*, among others. The start of the main chapter project consists of several buttons that will modify properties of the center movie clip. Figure 3-2 shows the layout of the file.

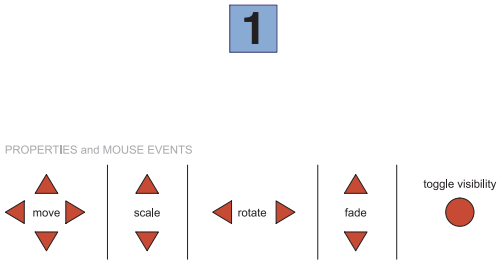

***Figure 3-2.** Layout of the* props_events.fla *file*

Starting with movement, we need to define one or more functions to update the location of the movie clip. There are two common approaches to this task. The first is to create one function for all movement that uses a conditional to decide how to react to each event. We'll demonstrate that when we discuss keyboard events. For now, we'll use the simpler direct approach of defining a separate basic function for each type of movement as shown in Example 3-1.

***Example 3-1.** props_events.fla*

```
1   function onMoveLeft(evt:MouseEvent):void {
2       box.x -= 20;
3   };
4   function onMoveRight(evt:MouseEvent):void {
5       box.x += 20;
6   };
7   function onMoveUp(evt:MouseEvent):void {
8       box.y -= 20;
9   };
10  function onMoveDown(evt:MouseEvent):void {
11      box.y += 20;
12  };
```

Once the functions are defined, all you have to do is add the listeners to the appropriate buttons.

```
13  move_left_btn.addEventListener(MouseEvent.MOUSE_UP, onMoveLeft);
14  move_right_btn.addEventListener(MouseEvent.MOUSE_UP, onMoveRight);
15  move_up_btn.addEventListener(MouseEvent.MOUSE_UP, onMoveUp);
16  move_down_btn.addEventListener(MouseEvent.MOUSE_UP, onMoveDown);
```

This simple process is then repeated for each of the buttons on stage. The remaining script collects the aforementioned properties and event listeners to complete the demo pictured in Figure 3-2.

```
17  scale_up_btn.addEventListener(MouseEvent.MOUSE_UP, onScaleUp);
18  scale_down_btn.addEventListener(MouseEvent.MOUSE_UP, onScaleDown);
19
20  rotate_left_btn.addEventListener(MouseEvent.MOUSE_UP, onRotateLeft);
21  rotate_right_btn.addEventListener(MouseEvent.MOUSE_UP,
    onRotateRight);
22
23  fade_in_btn.addEventListener(MouseEvent.MOUSE_UP, onFadeIn);
24  fade_out_btn.addEventListener(MouseEvent.MOUSE_UP, onFadeOut);
26
27  toggle_visible_btn.addEventListener(MouseEvent.MOUSE_UP,
    onToggleVisible);
28
29  function onScaleUp(evt:MouseEvent):void {
30      box.scaleX += 0.2;
31      box.scaleY += 0.2;
32  };
33  function onScaleDown(evt:MouseEvent):void {
34      box.scaleX -= 0.2;
35      box.scaleY -= 0.2;
36  };
37
38  function onRotateLeft(evt:MouseEvent):void {
39      box.rotation -= 20;
40
```

```
};
function onRotateRight(evt:MouseEvent):void {
    box.rotation += 20;
};

function onFadeIn(evt:MouseEvent):void {
    box.alpha += 0.2;
};
function onFadeOut(evt:MouseEvent):void {
    box.alpha -= 0.2;
};

function onToggleVisible(evt:MouseEvent):void {
    box.visible = !box.visible;
};
```

Methods

Methods, the verbs of the ActionScript language, instruct their respective objects to take action. For example, you can tell a movie clip to stop playing by using its **stop()** method. Like properties, methods appear consistently in the dot syntax that is the foundation of ActionScript, following the object calling the method. For example, if the movie clip "box" in the main timeline issues the **stop()** method, the syntax would appear like this:

```
box.stop();
```

Also like properties, most ActionScript classes have specific methods, and many inherit methods from ancestor classes. In addition, like properties, you can further define your own methods by writing functions in your own custom classes. For the following demonstration, we'll again focus on the movie clip from the prior example. This time, however, we'll introduce another event class and show you how to control your movie clips with the keyboard.

Using Keyboard Events to Call Methods

Trapping keyboard events is very similar to trapping mouse events, with one significant exception: The target of the event listener is not frequently the object you wish to manipulate. When working with text, the text field being manipulated may, indeed, serve well as the target of the keyboard events. When controlling movie clips, however, the stage itself is often a useful, centralized recipient of the keyboard events.

Adding an event listener to the stage means that you can process all key events with a single listener, and then isolate only the desired key events with a conditional, issuing instructions accordingly. To simplify the syntax of this last segment of our demonstration script, we'll use the *switch* form of conditional statements. The **switch** statement, reviewed in Chapter 2, is simply a more easily readable **if/else-if** conditional structure.

We'll start by adding the listener to the stage. In this case, we'll be looking for the **key down** event, which is specified using a constant like most predefined

events, but this time it is part of the **KeyboardEvent** class. When the event is heard, our listener will call the **onKeyPressed()** function.

```
stage.addEventListener(KeyboardEvent.KEY_DOWN, onKeyPressed);
```

Next, we define the **onKeyPressed()** function, being sure to type the incoming argument value as **KeyboardEvent**. Finally, we parse the **keyCode** property from the incoming event information now stored in the **evt** argument. The **keyCode** is a unique number assigned to each key and allows you to determine which key was pressed.

One **keyCode** value is assigned to each key, so this value can't be used directly for case-sensitive key checking—that is, uppercase "S" has the same **keyCode** as lowercase "s." If you need to analyze case sensitivity, use **charCode**, which does have unique values for each case.

To specify each key, we'll use constants defined in the **Keyboard** class, rather than having to remember each numeric **keyCode** value. This makes it easier to reference the Enter/Return key as **Keyboard.ENTER**, the left arrow key as **Keyboard.LEFT**, and so on.

We'll use five keys to call five methods. When each desired key is pressed, it will execute the appropriate method, and then break out of the switch statement. We'll also add a default state that will trace the keyCode of any other key pressed. The final script segment looks like this:

```
function onKeyPressed(evt:KeyboardEvent):void {
    switch (evt.keyCode) {
        case Keyboard.ENTER:
            box.play();
            break;
        case Keyboard.BACKSPACE:
            box.stop();
            break;
        case Keyboard.LEFT:
            box.prevFrame();
            break;
        case Keyboard.RIGHT:
            box.nextFrame();
            break;
        case Keyboard.SPACE:
            box.gotoAndStop(3);
            break;
        default:
            trace("keyCode:", evt.keyCode);
    }
};
```

The first four methods are basic movie clip navigation options, playing, stopping, or sending the movie clip to the previous or next frame in its timeline. The last method sends the movie clip to its third frame and then stops its playback. We'll look at these and other navigation options in greater detail in Chapter 5 when we discuss timeline control.

WARNING

Depending on your setup, many key events will not function properly in Flash when using the Control>Test Movie command. This is probably not an error but, instead, a result of Flash Player using keyboard shortcuts just like the Flash application does. To test your key events, simply use the Control>Disable Keyboard Shortcuts menu command to disable keyboard shortcuts in the Player (that is, after invoking Test Movie). Be sure to reenable the shortcuts, or you won't be able to use cmd+W (Mac) or Ctrl+W (Windows) to close the window, or use other familiar shortcuts. Alternatively, you can test the movie in a browser.

This code can be seen in the *methods_events.fla* file in the accompanying source code, as well as the combined file, *props_methods_events.fla*, which includes both the properties and methods examples in this chapter.

Event Propagation

So far in this chapter, we've been working with objects in the *display list*. We'll explain the display list in greater detail in the next chapter but, in essence, the display list contains all visual objects in your file. It includes the stage, any loaded SWFs, and any shapes, buttons, movie clips, and so on, down to the most deeply nested clip.

Objects in the display list are part of a special event flow often referred to as *event propagation*. When the target of certain events, including mouse and key events, is in the display list, the event is not dispatched directly to the event target. Instead, it is dispatched to the display list, and the event propagates from the top of the list down to the event target, and then bubbles (works its way) back up through the display list again.

Consider two movie clips (mc2 and mc3) within a movie clip (mc1) that is on the stage. Next, imagine that the target of the event is the nested movie clip, mc2. When the desired event occurs, it is not dispatched directly to mc2, but rather to the display list. First, the stage receives the event, then any relevant loaded SWFs (including the root timeline, in this example), then the parent movie clip, mc1, and then the target of the event, mc2. After the event is received by the target, it then propagates back up through the display list to mc2, root, and stage. Figure 3-3 depicts the process, showing a mouse event dispatched to the top of the display list, the stage, making it's way through the root timeline and parent movie clip until it reaches the event target, and then bubbling back up through the display list again.

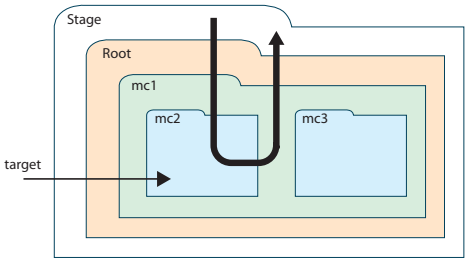

***Figure 3-3.** Event propagation process*

Event propagation can be used to great advantage with just a little bit of planning. For example, let's say both nested movie clips were designed to react to **mouse over** and **mouse out** events. Whenever the user rolled the mouse over one of the clips, it would change it **alpha** value to indicate interaction. In this case, you would normally have to attach a listener for each event to each movie clip. The code for such an example follows, and Figure 3-4 depicts the result, where each movie clip is represented by a folder: *folder0* and *folder1*. Example 3-2 shows the code in the sample file.

***Example 3-2.** event_propagational1.fla*

```
1   folder0.addEventListener(MouseEvent.MOUSE_OVER, onFolderOver);
2   folder0.addEventListener(MouseEvent.MOUSE_OUT, onFolderOut);
3   folder1.addEventListener(MouseEvent.MOUSE_OVER, onFolderOver);
4   folder1.addEventListener(MouseEvent.MOUSE_OUT, onFolderOut);
5
6   function onFolderOver(evt:MouseEvent):void {
7         evt.target.alpha = 0.5;
8   }
9
10  function onFolderOut(evt:MouseEvent):void {
11        evt.target.alpha = 1;
12  }
```

***Figure 3-4.** The effect of the changing alpha values using t* **mouse over** *and* **mouse out** *events*

Figure 3-4 represents the standard listener approach, in which listeners for **mouse over** and **mouse out** events are attached to both folders. As the mouse moves over a folder, the **alpha** value changes.

Now imagine having to use the same approach for many folders, as seen in Figure 3-5. The code could get quite extensive with all those listeners for each folder. However, with event propagation, it is possible to attach the listener to the parent movie clip, *folder_group* (indicated by the dashed line). The event will cascade through the display list, and the common listener functions will simply parse the object that is the intended target. The code that follows is significantly simplified thanks to event propagation.

```
1   folder_group.addEventListener(MouseEvent.MOUSE_OVER, onFolderOver);
2   folder_group.addEventListener(MouseEvent.MOUSE_OUT, onFolderOut);
3
4   function onFolderOver(evt:MouseEvent):void {
5         evt.target.alpha = 0.5;
6   }
7
8   function onFolderOut(evt:MouseEvent):void {
9         evt.target.alpha = 1;
10  }
```

Looking at Figure 3-5 again, the folders are numbered left to right, top to bottom, starting with 0. Imagine moving your mouse over *folder0*. The target of the event dispatched to the display list will be *folder0*, it will propagate through the list until it reaches *folder0*, and then it will bubble back up. Similarly, if you mouse over *folder5* or *folder10*, the listener function will know which folder was the target by parsing the **target** property of the event, and the alpha value of the appropriate folder will be changed. This can be

seen in the source file *event_propagation2.fla*. Figure 3-5 simulates listeners attached not to each folder, but rather to the parent movie clip (represented by the dashed line) within which each folder resides. Due to event propagation, the `mouse over` and `mouse out` events automatically dispatched to the display list are thereafter received by every child of the target movie clip.

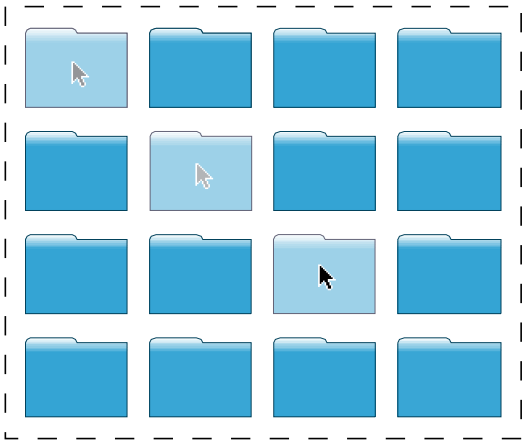

Figure 3-5. Using the parent movie clip to propagate events

WARNING

It is important to note that not all events propagate through the display list. Frame events, for example, which we'll discuss in the next section, are dispatched directly to the event target. Before relying on event propagation, check the documentation to see how the event behaves. In particular, the `bubbles` *property is a Boolean that indicates whether or not the event bubbles back up through the display list after reaching its target. For more information, see the companion web site, which includes discussions about event phases, priority of execution, stopping event propagation, and more.*

NOTE

Consult Essential ActionScript 3.0, Chapters 12 and 21, for more advanced discussions on event propagation.

Frame and Timer Events

We have been using mouse and keyboard events because you are almost certainly familiar with them to some degree, and they are ideally suited to this tutorial context. However, there are many, many events in the ActionScript language. While it's not possible to cover every one, we would like to round out the chapter with two significant other event types: *frame* and *timer*.

Frame Events

Frame events are not triggered by user input, the way mouse and keyboard events are. Instead, they occur naturally as the Flash file plays. Each time the playhead enters a frame, a frame script is executed. This means that frame scripts execute only once for the life of the frame, making them an excellent

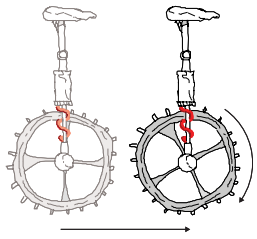

Figure 3-6. Visual depiction of the unicycle movements

NOTE

This example also demonstrates a scripting shortcut aided by ActionScript. When specifying a rotation higher than 360 degrees, ActionScript will understand and use the correct value—that is, 360 degrees is one full rotation around a circle, bringing you back to degree 0 (720 degrees is twice around the circle and also equates to 0). Similarly, 370 degrees is equivalent to 10 degrees, as it is 10 degrees past degree 0, and so on. This allows you to set the rotation of the wheel movie clip to the x coordinate of the mouse, without worrying about moving past the 360-pixel point on the stage.

location for seldom-executed tasks, such as initializations. In other words, for a frame script to execute more than once, the playhead must leave the frame and return—either because of an ActionScript navigation instruction, or a playback loop that returns the playhead to frame 1 when it reaches the end of the timeline.

However, using an event listener, you can listen for a recurring `enter frame` event that some display objects have, including the main timeline and the movie clips. An `enter frame event` is fired at the same pace as the document frame rate. For example, the default frame rate is 12 frames per second, so the default enter frame frequency is 12 times per second. Using the `enter frame` event allows your file to update frequently—a particularly handy thing for visual assets.

The *enter_frame.fla* file in the accompanying source code demonstrates this event by updating the position of a unicycle every enter frame. It places the unicycle at the location of the mouse and, to further review your work with properties, rotates the child movie clip in which the wheel resides. Figure 3-6 visualizes the effect. As you move your mouse to the right on the stage, the unicycle will move to the right, and the wheel will rotate clockwise.

The code for this example follows. The first line adds an enter frame event listener to the main timeline, specifying the event using the `ENTER_FRAME` constant of the `Event` class. The function sets the unicycle's x coordinate and rotation to the x coordinate of the mouse. This code can be found in the source file *frame_events.fla*.

```
1  stage.addEventListener(Event.ENTER_FRAME,onFrameLoop);
2
3  function onFrameLoop(evt:Event):void {
4      cycle.x = mouseX;
5      cycle.wheel.rotation = mouseX;
6  }
```

Timer Events

An alternative to using enter frame events to trigger actions on a recurring basis is to use time-based events. Although it's among the most straightforward options, using the `enter frame` event exclusively for this purpose has disadvantages. For example, Flash Player can only reliably achieve moderate frame rates—somewhere between the default 12 frames per second, and perhaps 18 to 25 fps on the high end. Your mileage may vary, but that's fairly accurate when averaging the CPU population at large. More importantly, the rate at which the enter frame fires is not always consistent.

On the other hand, time-based events are measured in milliseconds and, therefore, can fire more quickly. Further, time-based events don't vary from scenario to scenario, so they are more reliable and consistent.

Previous versions of ActionScript used the **setInterval()** method for ongoing recurring events, and the **setTimeout()** method for finitely recurring events. ActionScript 3.0 wraps up these approaches neatly behind the scenes of the new **Timer** class, simplifying the process of using timers.

The first step in using the **Timer** class is to create an instance of the class, as seen here:

```
var timer:Timer = new Timer(delay:Number, repeatCount:int);
```

The class constructor takes two arguments, the first is mandatory, and specifies the delay, in milliseconds, before the timer event is fired. The second parameter is optional and is the number of times the event fires. Omitting the second parameter will cause the event to fire infinitely, each time after the specified delay, similar to prior **setInterval()** implementations. Using a positive value, such as 1, will cause the event to fire that many times (again, after the specified delay), similar to prior **setTimeout()** implementations.

In the sample *timer_events.fla* in the accompanying source code, the timer event (specified as the constant **TIMER** in the **TimerEvent** class), occurs every second (1,000 milliseconds) and calls a function that adds rotation to a hand nested inside a watch movie clip. The code is quite simple, as shown in the following example:

```
1  var timer:Timer = new Timer(1000);
2  timer.addEventListener(TimerEvent.TIMER, onTimer);
3  timer.start();
4
5  function onTimer(evt:TimerEvent):void {
6      watch.hand.rotation +=5;
7  }
```

One important thing to note is line 3. The timer you instantiate does not start automatically the way prior intervals or timeouts started. This gives you greater flexibility and control over your timer events. You can also stop the timer using the **stop()** method, and reset the timer using the **reset()** method. The latter stops the timer and also resets the repeat count to zero. For example, if you specified that the timer call a function five times, but reset it after the third call, the timer would begin counting again from zero rather than picking up from three at the point when it was reset. Figure 3-7 depicts the code in the previous code sample. The hand on the stopwatch advances 5 degrees of rotation every second when a timer event calls a function.

NOTE

As described in Chapter 2, frame and timer loops, such as those seen in the previous examples, are often an attractive alternative to **for..loops** *because they allow additional updates to occur throughout the file. A code loop, such as a* **for..loop**, *is one of the most processor-intensive structures and will execute only the code inside the loop until the loop is finished. This means that other animation, sound or video updates, process progress reports, and so on, will all be halted while the loop is working.*

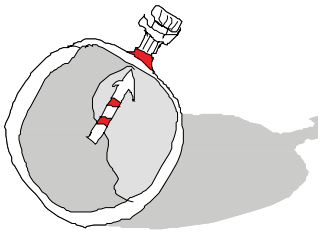

Figure 3-7. *Use of the timer event in a stopwatch*

Removing Event Listeners

While event listeners make most event handling easy to add and maintain, leaving them in place when unneeded can wreak havoc. From a logic standpoint, consider what could happen if you kept an unwanted listener in operation. Imagine a weeklong promotion for radio station 101 FM, which rewards customer number 101 who enters a store each day of that week. The manager of the store is set up to listen for "customer enter" events and, when customer 101 enters the store, oodles of prizes and cash are bestowed upon the lucky winner. Now imagine if you left that listener in place after the promo week was over. Oodles of prizes and cash would continue to be awarded at great, unexpected expense.

Unwanted events are not the only problem, however. Every listener created occupies a small amount of memory. Injudiciously creating many event listeners, without cleaning up after yourself, can result in a memory leak. Therefore, it's a good idea to remove listeners when you know they will no longer be needed.

To do so, you simply need to use the **removeEventListener()** method. By specifying the owner of the relevant event and the listener function that is triggered, you can remove that listener so it no longer reacts to future events. The **removeEventListener()** method requires two parameters: the event and function specified when the listener was created. Specifying the event *and* function is important because you may have multiple listeners set up for the same event.

Let's add to the previous example and remove the timer event listener when the rotation of the watch hand meets or exceeds 25 degrees of rotation. The new code is in bold.

```
1  var timer:Timer = new Timer(1000);
2  timer.addEventListener(TimerEvent.TIMER, onTimer);
3  timer.start();
4
5  function onTimer(evt:TimerEvent):void {
6      watch.hand.rotation +=5;
7          if (watch.hand.rotation >= 25) {
8                  timer.removeEventListener(TimerEvent.
   TIMER, onTimer);
9          }
10 }
```

As discussed earlier, this can be accomplished using a repeat count in the timer, like this:

```
var timer:Timer = new Timer(1000, 5);
```

However, the point of the example is to show you how to remove the listener from your logic flow and, equally important, from memory, when it is no longer needed. We briefly discuss an additional scenario for removing listeners in the "Garbage Collection" sidebar but, in all cases, it's good practice to remove any listeners that you know you'll no longer need. This is demonstrated in the source file *removing_listeners.fla*.

A RECOMMENDED OPTIONAL PARAMETER FOR EVENT LISTENERS

Garbage Collection

Garbage collection is the method by which Flash Player purges from memory objects that you no longer need. Garbage collection and memory management typically are not topics you need to concern yourself with when just getting started with ActionScript 3.0. However, there are some intermediate coding practices that you can adopt relatively painlessly—even at the outset of your learning—that may prove useful habits in the long run. Garbage collection is such a practice.

We just want to scratch the surface of this subject, laying the groundwork for conventions that we'll use throughout the remainder of this book, and then refer you to additional resources for more information.

There are three optional parameters that you can add to the end of the **`addEventListener()`** method. Here is the syntax of the method, with which you are probably already partly familiar if you've read this chapter. The optional parameters we'll discuss are in bold.

```
eventTarget.addEventListener(EventType.EVENT_NAME,
    eventResponse, useCapture:Boolean, priority:
    int, weakReference:Boolean);
```

The first two optional parameters control when the listener function executes. You probably won't need to adjust these values, but here's a quick snapshot of their functionality, so you can decide whether you want to explore them further.

The first optional parameter, **`useCapture`**, allows you to handle the listener event before it reaches its target (if set to **`true`**) or once the event has reached its target (if set to **`false`**) or is bubbling back up through the display list. The default (**`false`**) is to react to all events captured at or after the event reaches the target, and this is the configuration you will likely use most of the time.

The second optional parameter, **`priority`**, allows you to order the execution of multiple listeners set to respond to the same event in the same phase. This, too, is unlikely to be an issue, and the default parameter of **`0`** will serve you well in the vast majority of circumstances.

The third optional parameter, **`weakReference`**, is the option we want you to understand and start using. In a nutshell, this helps with memory management, in the event that you're not careful about removing unneeded listeners.

Briefly, in ActionScript 3.0, memory management that you do not explicitly control is handled behind the scenes by the garbage collector. When you are no longer referencing an object in your application, it is marked for cleanup, and the garbage collector periodically sweeps through your application discarding unneeded items, freeing up memory along the way. However, if a reference to an object remains, the garbage collector can't know that the object should be purged from memory.

Try as we might to be good, it's not uncommon for developers to forget to remove event listeners in their code (see the section "Removing Event Listeners" in this chapter). However, a distant next-best thing is a weakly referenced listener. Simply put, weakly referenced listeners aren't supervised by the garbage collector and, therefore, don't have to be manually marked for removal. If only weak references to an object remain after you have finished using it, then the object is eligible for collection.

Using this option is very simple. All you need to do is change the **`weakReference`** setting of the **`addEventListener()`** method from its default value of **`false`**, to **`true`**. Because it's the third optional parameter, values for the first and second parameters must be included so Flash knows which parameter you are trying to set. You will rarely need to change those values, so you can use their aforementioned defaults (**`false`** for **`useCapture`** and **`0`** for **`priority`**).

So, our preference, and the convention we will use hereafter in this book, is to use the **`addEventListener()`** method with this syntax:

```
eventTarget.addEventListener(EventType.EVENT_NAME,
    eventResponse, false, 0, true);
```

If you get in the habit of using this syntax, you will be far less likely to run into memory management problems due to lax code maintenance. Remember, this is not a substitute for removing your unneeded listeners explicitly. However, it's a backup plan, and a best practice that is easy to adopt.

Additional discussion of the event flow—including event phases, setting listener priority, stopping propagation along the way, manually dispatching events, and more—is featured on the companion web site. Flash developer Grant Skinner also wrote a helpful series of articles on resource management on his blog (*http://www.gskinner.com/blog*) that got us thinking about this in the first place. Finally, event flow is discussed in depth in Chapters 12 and 21 of our resource book of choice, *Essential ActionScript 3.0*.

What's Next?

This chapter has demonstrated several ways to manipulate Flash objects but in the case of our example movie clip, has assumed that the movie clip already existed on the stage. This is an acceptable assumption for projects authored primarily using the timeline, but it is a limiting assumption. If all files are to be constrained by using only elements manually added to the stage at author time, and used only in the manner and order in which they were originally added, the files cannot be as dynamic as the ActionScript language allows.

Coming up, we'll talk more about the display list—an excellent means of managing visual assets. Understanding the basics of the display list is instrumental not only in dynamically adding elements at runtime, but also manipulating existing stage-bound objects to their fullest potential.

In the next chapter, we'll discuss:

- Adding new children to the display list
- Removing existing display list children
- Swapping depths of objects in the display list to change their visual stacking order dynamically
- Managing the hierarchical relationship of display list objects, and how to change that relationship through reparenting