Part III

Dynamic Visuals

Chapter 7

Motion, Tweening, and Easing

Things moving. Isn't that what drew most of us to Flash? Motion is deep in Flash's genetic code. The Timeline animation tools haven't changed much over the various versions of Flash. But in Flash 4 and 5, and now MX, the growing capabilities of ActionScript have provided powerful and creative new ways of making things move. In this chapter, we'll explore the concepts of motion, tweening, and easing. We'll develop a thorough understanding of these ideas, and then look at how to realize them in ActionScript. That said, it's time to get in touch with our motions.

Concepts of Motion

I have spent quite some time thinking about motion as it applies to programmatic animation. This is something that comes from my philosophy training—analyzing a concept to break it into definitive components. More specifically, the issue is this: what constitutes *a motion*—an identifiable instance of motion—from a programmer's point of view? To keep it simple, I decided to restrict my consideration to *one-dimensional* motion, at least initially. I came up with this definition: a motion is *a numerical change in position over time*. Breaking this down even further, we see that there are three key components of a motion: position, time, and beginning position.

- **Position** A motion has a unique numerical value—its *position*—at any point in time. When the value of the position changes, there is movement.
- **Time** A motion occurs over *time*. Typically, time is measured in positive numbers. In my mind, it makes sense to standardize motion time by starting it at zero, like a shuttle launch. After that, a motion's time property ticks away like its own personal stopwatch.

- **Beginning Position** A motion has to start somewhere. From our perspective, the motion starts at the moment in time we start observing it—in other words, when time is zero. Thus, we define *beginning position* as the motion's position when time is zero.

More on Position

By *position*, I don't mean merely *physical* position. For our purposes, *position* can be any numerical quantity. It could be something visual like the movie clip properties `_x`, `_xscale`, or `_alpha`. It could also be a variable representing money, temperature, music volume, or population density.

To be more specific, the *position* quantity should be a real number that is *one-dimensional*. In other words, position has a *scalar,* not a *vector* value, as discussed in Chapter 4. The number can be positive, negative, or zero, a decimal or an integer. In visual terms, we should be able to locate the quantity on a number line, either to the left or right of zero, or at zero itself. Figure 7-1 shows two positions on a number line, at –2 and 1.

NOTE: In contrast to real numbers, there are *imaginary numbers*, which essentially consist of a real number multiplied by *i*, the square root of -1. There are also *complex numbers*, which are two-dimensional. Each complex number is the sum of a real and an imaginary number.

Of course, we find motions in higher dimensions. We are most familiar with two- and three-dimensional movement in our world. However, a multi-dimensional position can usually be broken down into several one-dimensional positions. The Flash stage is a two-dimensional space, through which movie clips travel. A simple diagonal tween may look like one motion, but really it's a combination of changes in `_x` and `_y`, which are independent from each other. By contrast, `_width` and `_xscale` are not independent—changing one affects the other. Thus, they are not separate dimensions, but different aspects of the same dimension.

NOTE: Consider the fact that a movie clip has many numerical visual properties: `_x`, `_y`, `_xscale`, `_yscale`, `_rotation`, and `_alpha`. Consequently, you could say that an animated movie clip is moving through *six-dimensional space*. If you're interested in learning more about higher dimensions, I recommend the books *Flatland* and *The Mathematical Tourist*.

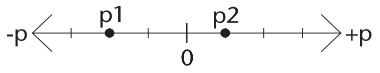

FIGURE 7-1

Positions p1 and p2 on a number line

It's important to realize that, from a programming perspective, the position of a motion is numerical *by necessity*. Any quantity that can be said to be *moving*, you should be able to represent with a number. For instance, suppose a string variable contains a single letter, and is "moving" through the letters of the alphabet—first it's "A," a second later it's "B," and so on. The quantity in question, letters, is not natively numeric. However, we know that there is a specific order to the symbols: B follows A as surely as 2 follows 1. It is possible to place the letters in order on a number line. We can easily correlate the letters of the alphabet with numbers; in fact, the ASCII codes have already done this. Thus, we can treat the changing letters as a numerical motion.

Position as a Function of Time

The aspect of time is crucial to motion—things change *over time*. Nothing can move in "zero time," or be in two places at once (although quantum theory may have some strange exceptions to this rule). In other words, a position needs time to change, and it can have only one value at a specific point in time.

Because position and time have this one-to-one relationship, we can say that *position is a function of time*. This means that, given a specific point in time, we can find one, and only one, corresponding position. In mathematical terms:

where p = position, t = time:

$p = f(t)$

In ActionScript, we can implement mathematical functions literally with—you guessed it—functions. Notice the similarity between the following code and the preceding math notation:

```
f = function (time) {
    return 2 * time;
};
// test f()
t = 12;
p = f(t);
trace (p); // output: 24
```

In comes the time, out goes the position. Since an ActionScript function can only return one value, we can see the importance of the one-to-one relationship between time and position. Later in the chapter, we'll look at my tweening functions, which provide meaningful and predictable relationships between position and time.

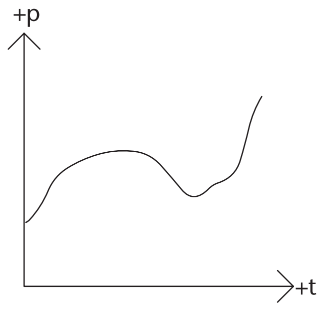

FIGURE 7-2

Graph of a motion's position over time

NOTE: In a sense, a function can have many values at once if you pack an object with properties and return the object. However, what is returned from the function is still one value—an object reference.

Motion as a Graph

Time is one-dimensional, and for this discussion, we are keeping position one-dimensional as well. Position can be represented on a number line, and time can be displayed on another number line. If we put these two number lines at right angles to each other, we have a Cartesian graph (discussed in Chapter 3). The standard practice is to put *time* along the x-axis, and the *function of time* (*position* in our case) on the y-axis.

With this setup, we can visually represent a motion by plotting points on the graph. Instead of (x, y), we have (p, t) coordinates. Figure 7-2 shows a graph of a possible motion.

Static Tweens in Flash

The word "tween" is derived from "between." The term comes from traditional animation, where the work is split between keyframes and tween frames. Some of the animators would draw the keyframes, and others, perhaps those with less talent or seniority, would do the grunt work of filling in the gaps with tween frames.

The concept transfers well to Flash. On the Timeline, a designer can define keyframes, and tell Flash to do the mundane work of tweening between them. I call these *Timeline* tweens. You define them at author-time, on the Timeline, and Flash "hard-codes" that motion into the SWF. You can't change these tweens at run-time, so I also call them *static* or *author-time* tweens. The converse would be *dynamic*, *run-time*, *ActionScript* tweens, which we'll explore later in the chapter.

You may be interested to know that author-time tweens actually don't exist in a SWF file. I once thought that Flash stores the keyframes in the SWF and then calculates the tween on the fly. I discovered, however, that this is not the case. When you publish your movie, Flash simply keyframes the entire tween. You can verify this by comparing the file sizes of SWFs with the same movement, one with a tween, the other with keyframes. Try an experiment by following these steps:

1. Start with a new movie.
2. Create an empty movie clip symbol.
3. Drag the movie clip from the library to the stage to create a keyframe on frame 1.
4. Insert a keyframe on frame 5.
5. Move the movie clip to a new position on frame 5.
6. Create a Timeline tween between frames 1 and 5.
7. Turn SWF compression off in the Publish Settings.
8. Test the movie (CTRL-ENTER). The SWF should be about 104 bytes.
9. Now keyframe the entire tween by selecting all of the frames and pressing F6.
10. Test the movie. The SWF has the same file size.

As you can see, author-time tweens have no advantage, in terms of file size, over keyframes. There is also no difference in processor usage between timeline tweens and keyframes.

Shape tweens are a different story. Through testing, I found that the transition frames in a shape tween *are* calculated on-the-fly—not pre-rendered in the SWF. As a result, you can increase the length (in frames) of a shape tween without seeing much increase in file size. If you convert a shape tween to keyframes, however, you will likely see an increase in file size if the shapes are sufficiently complex. I ran a test on a shape tween between two dense scribble shapes. With ten frames in the tween, the SWF file size was 38 KB. However, after keyframing the ten frames and clearing the shape tween setting, the same animation became 258 KB! The other end of the stick is that shape tweens are more demanding on the CPU than are motion tweens, as the intermediary curves must first be calculated and then rendered.

Dynamic Tweening in ActionScript

I have often heard ActionScript programmers express their desire to "break free of the Timeline," especially when it comes to animation. Timeline tweens are static and stolid. You're stuck with whatever endpoints and easing you define in the authoring tool.

Once you begin working with ActionScripted motion, however, a whole world of possibilities opens up. You can produce interactive, dynamic motion that seems smoother and more alive than Timeline tweens. Often, this involves using principles of physics to create bounce and acceleration, which we'll discuss in Chapter 8. In the remainder of this chapter, though, we'll explore a variety of ways to dynamically tween in ActionScript.

The Standard Exponential Slide

The most well-known technique of ActionScripted tweening is what I call the *standard exponential slide*. (It's tempting to coin a tacky acronym like *SES*, but I'll abstain.) The slide is a movement towards a destination that gradually slows to a stop—an ease-out. The slide is exponential because the speed can be represented with an equation that depends on a changing exponent. Later in the chapter, we will look at this equation. You may recognize the slide technique in the following code:

```
// a standard exponential slide
this.onEnterFrame = function () {
    var distance = this.destinationX - this._x;
    this._x += distance / 2;
};
```

At my first FlashForward conference, I remember hearing the concept of this slide explained by three different speakers. The idea is simply that you move a fraction of the distance—say one-half—to your destination in each unit of time. For instance, if you need to move a total of ten feet, you could move five feet, then 2.5 feet, then 1.25, then .625, and so on.

The ancient Greek philosopher Zeno explored this idea in one of his famous paradoxes: if you have to move halfway each time, you never actually reach your destination. (Zeno had a decent point, at least until quantum theory showed that space is not infinitely divisible.) In Flash, though, you soon get "close enough," that is, within a pixel of your destination, or until you've exceeded the available decimal places of floating-point numbers (at which point 9.9999… turns into 10).

All in all, the standard exponential slide is a dynamic ease-out motion easily generated with a tiny amount of ActionScript. This, of course, is why you see it everywhere. However, I grew dissatisfied with this technique, for several reasons. First of all, you can only do ease-out, not ease-in. Also, the motion is aesthetically different from Flash's own eased tweens. Compared to a Timeline tween with ease-out, the slide moves "too fast" in the beginning, then "too slow" towards the end.

Furthermore, you can't directly find the tween's position for a given time. For instance, you have no way of knowing exactly where the moving object will be when ten frames have elapsed. Each frame's position is dependent on the previous frame. Consequently, you have to run the whole process from the beginning to get to a specific point in time. There's no easy way to jump to an arbitrary frame in the tween.

Lastly, you can't directly specify the duration of the tween. The formula doesn't let you make the tween, say, 20 frames in length. Basically, "it gets there when it gets there." You can change the acceleration factor to speed up the process or slow it down, but it takes trial and error to get a tween of the desired duration.

Noticing these shortcomings, I made it my mission to find an alternative to the standard exponential slide. I wanted tweens and easing that would give more control to the ActionScript developer. But first, more analysis was needed.

Key Components of a Tween

A tween can be defined as *an interpolation from one position to another*. Thus, a tween fits our earlier definition of a motion—a change in position over time. Tweens are a subset of motions, just as diamonds are a subset of gems. As such, tweens and motions share the three essentials of *position*, *time*, and *beginning position*. In addition, tweens have two other essential characteristics: *duration* and *final position*.

Duration

A tween has to end. It has to take "less than an eternity" to reach its destination. In other words, a tween must have a *duration* that is less than infinity, but greater than zero.

NOTE: By contrast, a generic motion doesn't necessarily have a finite duration. For instance, a circular orbit is a motion that never ends. A motion without a duration is definitely not a tween.

Final Position

A tween fills in the gap between two keyframe positions, getting you "from Point A to Point B." Point A is the *beginning position*—something all motions have. Point B, meanwhile, is the *final position*, an essential component of a tween.

NOTE: It's also possible to replace final position with *change in position*, which is simply the difference between Point A and Point B. For example, you can either say, "start at 10 and go to 15," or you can say, "start at 10 and go forward by 5." Final position is an *absolute* coordinate, whereas change in position is a *relative* coordinate.

Tweening Functions

Naturally, there are different ways to implement tweening in ActionScript. However, having boiled tweens down to their essence, I believe that my code captures the key elements—no more, no less.

The purpose of a tweening function is to *deliver a position for a specific time, given the tween's essential characteristics*. These characteristics are beginning position, final position, and duration.

In terms of structure, we have five quantities to incorporate into the function. Position is the one that is spit out the back end. The other four items are fed to the function as parameters. A tweening function should have this approximate structure, then:

```
getTweenPosition = function (time, begin, final,
duration) {
    // calculate position here
    return position;
};
```

Alternatively, *change in position* could be used instead of *final position*, in which case, the function would look like this:

```
getTweenPosition = function (time, begin, change,
duration) {
    // calculate position here
    return position;
};
```

With my tweening functions, I found that I could optimize the code a bit more if I used *change*. I also decided to store the functions in the *Math* object, to make them globally accessible. My tweening functions have this format:

```
Math.tweenFunc = function (t, b, c, d) {
    // calculate and return position
};
```

NOTE: I shortened the parameter names to one letter for speed. Variables with longer names take longer to look up than variables with short names, because of the interpreter's internal hashing procedure. I heard this from Gary Grossman, the creator of ActionScript. Timing tests run by members of the Flashcoders list have also confirmed the difference in speed.

As for the actual calculation of the tween's position—that's where things get interesting. We are now ready to explore specific equations that interpolate between two points in diverse ways.

Linear Tweening

"The shortest path between two points is a straight line"—so goes the saying. When you want to move from Point A to Point B, a linear tween is the simplest path, mathematically speaking. This tween has constant velocity throughout; there is no acceleration. In Flash, a Timeline tween with an easing of zero is a linear tween.

Graph

When you graph a linear tween over time, it comes out as a straight line. That's why I call it a "linear" tween. Figure 7-3 shows the graph of a linear tween's position over time.

In this chapter, the tweening graphs I use are all "normalized" in a particular way. Both axes—*position* and *time*—are plotted between standard values of 0 and 1. Of course, in real-world situations, position and time range between many different values. For instance, a tween may move a total of 120 pixels in 30 frames. If we graphed this motion, the position axis would be displayed between 0 and 120, and time would range from 0 to 30. However, we can easily stretch our original normalized (0 to 1) graph to match this new graph. All it takes is simple multiplication: 30 horizontally by 120 vertically.

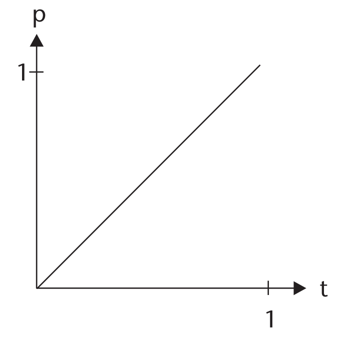

FIGURE 7-3

Graph of a linear tween's position over time

The equation for the graph in Figure 7-3 is quite simple:

$p(t) = t$

This is the basic $y = x$ equation for a diagonal line that most of us learned in high school math. If the time is 0.6, the position is also 0.6.

As the graph stands, the slope of the line (rise over run) is 1: position (the rise) changes by 1 over the course of the tween, and so does time (the run). This is the normalized form of the equation, though, with values ranging from 0 to 1. If we take the example tween mentioned earlier, where position changes by 120, and the duration is 30, the slope is different:

slope = rise / run

slope = change in position / duration

slope = 120 pixels / 30 frames

slope = 4 pixels / frame

In this case, the slope is 4 pixels per frame. The slope of the line is also the velocity of the motion. A different tween covering 240 pixels in 30 frames would have a velocity of 8 pixels per frame. It moves faster than the first tween and its graph has a steeper slope.

We can generalize these results to come up with a standard equation for a linear tween, given the change in position and duration. We know that that velocity is constant during a linear tween, just as the slope is constant. Thus, we can find the tween's position at a particular time by simply multiplying velocity by time:

$p(t) = t \times v$

To take a real-life example, suppose your car's velocity is 60 miles per hour:

$v = 60$

If you've been driving two hours, how many miles have you covered? This is easy to work out in your head, but let's do the math explicitly:

$t = 2$
$p(t) = t \times v$
$p(2) = 2 \times 60$
$p(2) = 120$

We know that velocity equals change over duration, so we can replace *v* with *c* divided by *d*:

let c = change in position and d = duration
$v = c / d$
$p(t) = t \times v$
$p(t) = t \times (c / d)$

One last thing: remember that a tween has a beginning position. So far, we've assumed that the beginning position is zero. However, this often isn't the case. In terms of the car example, suppose you start 10 miles out of town, then drive at 60 mph for 2 hours. You'll end up 130 miles from the town (120 + 10).

We can incorporate the beginning position into our equation, simply by adding it at the end:

let b = beginning position
$p(t) = t \times (c / d) + b$

NOTE: The preceding equation is a form of the "slope-y-intercept" equation for a line: $y = mx + b$. Here, the variable m is the slope, and b is the y-intercept, the point where the line crosses the y-axis.

ActionScript Function

The code for our linear tween function follows the earlier equation closely:

```
Math.linearTween = function (t, b, c, d) {
    return c*t/d + b;
};
```

Four parameters enter the function—time, beginning position, change in position, and duration—and the corresponding position is returned.

Implementing a Tween with a Function

How do we actually use a tweening function? It's a matter of setting up the key properties of a tween, and feeding them into the function. First, we decide which property we want to set in motion. The `_x` property of a movie clip will do nicely to start. Tweening `_x` will produce a horizontal motion.

Then we need to choose the parameters of the tween. Where will it start, where will it finish, and how long will it take? We then establish these quantities in `begin`, `finish`, and `duration` properties inside the movie clip:

```
this.begin = 100;
this.finish = 220;
this.duration = 30;
```

This tween will start at 100 pixels and end up at 220, over the course of 30 frames. Looking ahead, we know that the tweening function requires the *change* in position, rather than the *final* position itself. So we might as well add a `change` property:

```
this.change = this.finish - this.begin;
```

A tween's time starts at zero, so let's initialize it as another property:

```
this.time = 0;
```

To produce animation, we need a frame loop. This calls for an *onEnterFrame( )* handler for the movie clip:

```
this.onEnterFrame = function () {
    // do tweening stuff
};
```

Now we have to figure out what code to put in the frame loop. What properties change over the course of the tween? Our first three properties, `begin`, `change`, and `duration`, don't vary. It is `time` and `position` that change during the tween. We'll have `time` track the number of frames that have elapsed. Thus, it should be incremented every frame:

```
this.onEnterFrame = function () {
    this.time++;
};
```

The other changing property is position. How do we get the position as time flies by? We call upon our tweening function, which takes four tween parameters and returns the position. So we could call it like this:

```
this.position = Math.linearTween (this.time, this.begin,
    this.change, this.duration);
this._x = this.position;
```

We can make this more straightforward by assigning the `position` value directly to the `_x` property:

```
this._x = Math.linearTween (this.time, this.begin,
    this.change, this.duration);
```

We can shorten this even further by using the `with` statement:

```
with (this) {
    _x = Math.linearTween (time, begin, change,
duration);
}
```

Setting the scope to `this` eliminates the need to reference each of the properties with the "this" prefix.

Now we can put the tweening function inside the *onEnterFrame( )* handler to produce animation, with the following code:

```
this.onEnterFrame = function () {
    with (this) {
        _x = Math.linearTween (time, begin, change, duration);
    }
    this.time++;
};
```

The position is calculated, then time is incremented by one. We can simplify a bit by incremented time in the same step as passing it to the function:

```
this.onEnterFrame = function () {
    with (this) {
        _x = Math.linearTween (time++, begin, change, duration);
    }
};
```

Putting all of our code together, we have this:

```
this.begin = 100;
this.finish = 220;
this.change = this.finish - this.begin;
this.duration = 30;
this.time = 0;
this.onEnterFrame = function () {
    with (this) {
        _x = Math.linearTween (time++, begin, change, duration);
    }
};
```

At this point, you can stick this code into a movie clip and try it out. It should work—for the most part. There will be one small problem, however: the tween won't end. For the animation to work, we need to stop when the tween's time is up. If the time is greater than the duration, we should stop the frame loop by deleting the *onEnterFrame( )* handler. By adding one line of code to perform this check, our tween is finished:

```
// a linear tween in _x
this.begin = 100;
this.finish = 220;
this.change = this.finish - this.begin;
this.duration = 30;
this.time = 0;
this.onEnterFrame = function () {
    with (this) {
        _x = Math.linearTween (time++, begin, change, duration);
        if (time > duration) delete this.onEnterFrame;
    }
};
```

In this code, you can replace `Math.linearTween` with another tweening function to produce a different style of tween. Thankfully, all my tweening functions have the same structure, making them easily interchangeable. We'll look at the different flavors of tweening and easing later in the chapter. But first, let's return to our discussion of linear motion.

Aesthetics of Linear Motion

I'll be honest—I don't care much for linear tweens. They look stiff, artificial, mechanical. You mostly see unaccelerated motion in machinery. An automated factory is chock-full of linear tweens—robotic arms, conveyor belts, assembly lines. It is extremely difficult for humans to move in a linear fashion. Breakdancers who do "robot"-style moves have achieved their linearity with much practice. In this sense, you could say Michael Jackson is the master of the linear tween.

So, there are some legitimate uses of linear tweens. If you're animating robots or breakdancers, they're perfect. For most other things, though, linear motion can look downright ugly. I have often come across animations where the author obviously threw down some keyframes and tweened between them without applying any easing. It drives me nuts.

I don't know how much the average person picks up on the naturalness or falsity of motion in Flash movies, but in 3-D animation, people certainly are quick to say when the movement "looks totally fake," even when the object texture is photorealistic. For example, in *Star Wars Episode II: Attack of the Clones*, there is some excruciatingly bad animation in the scene where Anakin is trying to ride one of the "Naboo cows." It's something most adults would notice, I think.

I have a theory that the mind is constantly interpreting an object's motion in order to *infer the forces* acting on it. When an object accelerates in an unnatural way, the brain picks up on it and says, "that's not possible with real-world forces." In real life, suppose you saw a ball roll to the right at a constant speed, then abruptly roll to the left at a faster speed. You would think to yourself, "Something must have hit it"—*even if you didn't see a collision*. Similarly, when an object switches from one linear tween to another, there is an instantaneous change in velocity, like the ball changing direction suddenly. When I see this, I *feel* like the object has been jerked by an invisible chain, or bounced off an unseen wall. But when I can't see a chain or a wall, the "cognitive dissonance" drives me crazy. "It's not supposed to move that way!" I protest. It takes some easing to calm me down.

Easing

Easing is acceleration, a change in speed. In this chapter, I focus on easing as the transition between the states of *moving* and *not-moving*. When an object is moving, it has a velocity of some magnitude. When the object is not moving, its velocity is zero. How does the velocity change—what does the acceleration look like?

In the real world, velocity doesn't suddenly jump from one value to another. Sometimes it looks that way, for instance, when you hit a baseball with a bat. Even here, though, the change isn't instantaneous. If you watch the collision in super-slow motion, you can see the ball gradually slowing to a stop, and actually deforming around the bat. Then, the ball accelerates in the direction of the bat's swing and flies off. There is a deceleration to a speed of zero, then an acceleration in the other direction—two instances of easing.

Aesthetics of Eased Motion

Most movements in nature have some kind of easing. Organic processes typically involve forces (which we'll discuss in detail in Chapter 8). If the forces aren't in balance, acceleration is produced. Our minds make this connection between acceleration and force intuitively. When we see the velocity of an object gradually changing, we infer that some force is steadily pushing or pulling it. As a result, animation usually looks more natural and fluid when it follows similar rules.

Elevators are great examples of the importance of easing. Imagine riding in an elevator that didn't start or stop moving with any easing. You press the button to go to the twentieth floor. The elevator doors close, then the whole thing suddenly jerks upwards, throwing you to the floor. Getting back to your feet, you watch the numbers: eighteen… nineteen… *ding!* Your body flies straight upwards. As the doors open, you frantically try to pull your head out of the ceiling. Too late! The elevator flies back down to the first floor, and suddenly you're weightless.

This is a facetious example, to be sure. But I literally feel "jerked around" in a similar way when I watch animation without proper easing. Let's look at three general categories of easing: ease-in, ease-out, and ease-in-out.

Ease-In

Start slow and speed up—that's an *ease-in*. With author-time tweens, an easing value of -100 produces an ease-in. In my Flash 4 days, when I majored in Timeline tweens, I nearly always used easing values of either 100 or -100. In general, I don't find less-than-100 percent easing very useful. Therefore, from this point on, I will use "easing" to mean 100 percent easing.

If we graph the position of an ease-in tween over time, we arrive at a curve like the one in Figure 7-4. With easing, we get a curve. Without easing, we get a line as in Figure 7-3.

Remember that the slope of a graph corresponds to velocity. The tween curve starts out horizontal—a slope of zero. Thus, at t=0, the velocity is

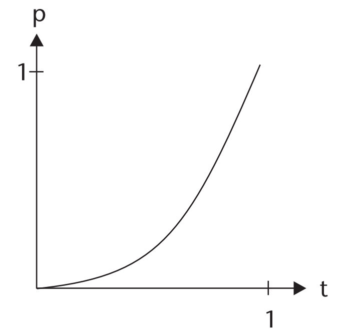

FIGURE 7-4

Graph of an ease-in tween

zero—a dead stop. Over time, the graph gradually becomes steeper and the velocity increases.

NOTE: Ease-in and ease-out are reversed in Flash from what they are in many other animation programs. In 3-D animation software, for instance, an ease-in slows down at the end of the tween, not the beginning.

Ease-Out

The inverse of an ease-in is an *ease-out*, where the motion starts fast and slows to a stop. A Timeline ease-out tween has an easing value of 100. Take a look at the ease-out curve in Figure 7-5.

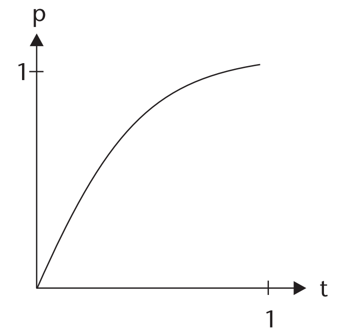

FIGURE 7-5

Graph of an ease-out tween

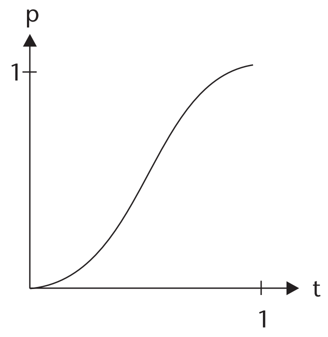

FIGURE 7-6

Graph of an ease-in-out tween

The graph starts out steep but levels off, becoming horizontal at the end of the tween. The velocity starts out at some positive value and steadily decreases to zero.

Ease-In-Out

The third category of easing is my favorite. An ease-in-out is a delicious half-and-half combination, like a vanilla-chocolate swirl ice cream cone. The first half of the tween is an ease-in; the second half, an ease-out. The curve of the ease-in-out, shown in Figure 7-6, is quite lovely (and bears more than a passing resemblance to a certain product logo).

I use in-out easing pervasively because it produces the most natural-looking motion. Think about how a real-world object might move from one point of rest to another. The object accelerates from a standstill, then slows down and comes to a stop at its destination. Elevators, for example, use in-out easing.

Unfortunately, Flash doesn't have in-out easing for author-time tweens. When I do Timeline animation, I am forced to create each ease-in-out with two tweens. As you can imagine, this is a real pain to maintain when I have to modify the animation. Every time I change an endpoint of the overall tween, I have to redo the two half-tweens. I had hoped to find an ease-in-out option in Flash MX, but alas, it was not to be.

Varieties of Eased Tweens

Thankfully, with ActionScripted motion, the possibilities are endless. With the right mathematics and code, we can define all manner of tweens with ease (pun intended).

Quadratic Easing

Flash's Timeline tweens use something called *quadratic easing*—which could actually be termed "normal" easing. The word *quadratic* refers to the fact that the equation for this motion is based on a squared variable, in this case, t^2:

$$p(t) = t^2$$

NOTE: I always wondered why the term *quad*-ratic (the prefix means "four") is used to describe equations with a degree of two (x^2). While writing this chapter, I finally looked it up in the dictionary (RTFD, you might say). I discovered that *quad* originally referred to the four sides of a square. Thus, a *squared* variable is *quadratic*.

I used the quadratic easing curve earlier in Figure 7-4. It's actually half a parabola. Here it is again, for reference purposes, in Figure 7-7.

Here's the quadratic ease-in ActionScript function:

```
Math.easeInQuad = function (t, b, c, d) {
    return c*(t/=d)*t + b;
};
```

Recall that `t` is time, `b` is beginning position, `c` is the total change in position, and `d` is the duration of the tween.

This equation is more complex than the linear tween, but it's the simplest of the equations that implement easing. Basically, I normalize `t` by dividing it by `d`. This forces `t` to fall between 0 and 1. I multiply `t` by itself to produce quadratic curvature in the values. Then I scale the value from a

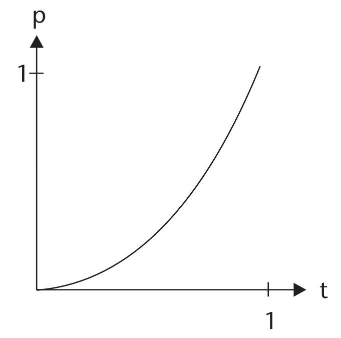

FIGURE 7-7

Graph of quadratic easing

normalized one to the desired output, by multiplying by `c`. I finish off the position calculation by adding the initial offset `b`, then returning the value.

NOTE: The `t/=d` bit in the preceding code is an optimization technique. This compound operator lets you divide and reassign `t`, *and* use its value in further operations, all in one step. The Flash Player uses a stack-based virtual machine that has four "registers" or memory locations reserved for holding data temporarily. Information can be stored and retrieved much faster from a register than from a variable, but registers are rarely used in Flash-generated bytecode. However, in-line syntax like `p = (t/=d) * t` is compiled to bytecodes that use one of the registers to temporarily store a value during the calculation, speeding up the process. (A special thanks goes to Tatsuo Kato for first applying this technique to my code.)

The following code is the quadratic ease-out function:

```
Math.easeOutQuad = function (t, b, c, d) {
    return -c * (t/=d)*(t-2) + b;
};
```

The original quadratic curve needed to be massaged a bit to get it where I wanted it. I multiplied *c* by -1 to flip the curve vertically. I also had to play with the value of *t* to shift the curve into place.

Now, here's the quadratic ease-in-out function:

```
Math.easeInOutQuad = function (t, b, c, d) {
    if ((t/=d/2) < 1) return c/2*t*t + b;
    return -c/2 * ((--t)*(t-2) - 1) + b;
};
```

I've combined the in and out code into one function—two half-tweens fused together. You may notice several divisions by 2 in the code. I did this to scale the equations to half their normal size, since each equation covers half of the time span. The ease-in equation governs the tween until half the time has elapsed, after which the ease-out equation takes over. Between the equation-switching and some additional curve-shifting, the code became increasingly cryptic—but beautifully so.

Cubic Easing

A cubic equation is based on the power of three.

$$p(t) = t^3$$

A cubic ease is a bit more curved than a quadratic one. Figure 7-8 shows the graph of a tween with a cubic ease-in.

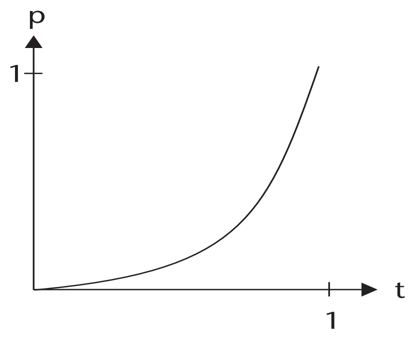

FIGURE 7-8

Graph of cubic easing

The following displays the cubic easing functions:

```
Math.easeInCubic = function (t, b, c, d) {
    return c * Math.pow (t/d, 3) + b;
};

Math.easeOutCubic = function (t, b, c, d) {
    return c * (Math.pow (t/d-1, 3) + 1) + b;
};

Math.easeInOutCubic = function (t, b, c, d) {
    if ((t/=d/2) < 1)
        return c/2 * Math.pow (t, 3) + b;
    return c/2 * (Math.pow (t-2, 3) + 2) + b;
};
```

You'll notice I used the *Math.pow( )* function here to raise numbers to the third power. In the first cubic easing function, for instance, I calculated $(t/d)^3$ like this:

```
Math.pow (t/d, 3)
```

Alternatively, I could cube this quantity by multiplying it by itself:

```
(t/=d)*t*t
```

However, once you get into higher exponents, it's faster to just call *Math.pow( )* to perform the multiplication.

Quartic Easing

A quartic equation is based on the power of four:

$$p(t) = t^4$$

The quartic ease, shown in Figure 7-9, puts just a bit more bend in the curve. A cubic ease, though more pronounced than a quadratic ease, still feels fairly natural. It's at the quartic level that the motion starts to feel a bit "other-worldly," as the acceleration becomes more exaggerated.

Here are the quartic easing functions:

```
Math.easeInQuart = function (t, b, c, d) {
    return c * Math.pow (t/d, 4) + b;
};

Math.easeOutQuart = function (t, b, c, d) {
    return -c * (Math.pow (t/d-1, 4) - 1) + b;
};

Math.easeInOutQuart = function (t, b, c, d) {
    if ((t/=d/2) < 1)
        return c/2 * Math.pow (t, 4) + b;
    return -c/2 * (Math.pow (t-2, 4) - 2) + b;
};
```

The code is similar in structure to the cubic functions, only with *Math.pow( )* raising `t` to the fourth power now and some adjusted curve shifting.

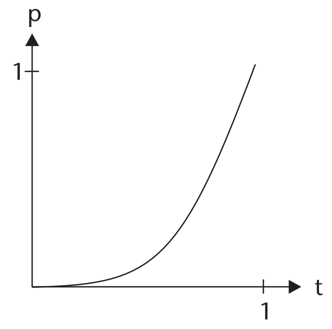

FIGURE 7-9

Graph of quartic easing

Quintic Easing

Quintic easing continues the upward trend, raises time to the fifth power:

$$p(t) = t^5$$

Quintic is a fairly pronounced curve, as Figure 7-10 shows. The motion starts out quite slow, then becomes quite fast. The curvature of the graph is close to that of exponential easing, discussed later in the chapter.

Putting all of the t^n ease curves on the same graph makes for an interesting comparison, as shown in Figure 7-11.

Here are the quintic easing functions:

```
Math.easeInQuint = function (t, b, c, d) {
    return c * Math.pow (t/d, 5) + b;
};

Math.easeOutQuint = function (t, b, c, d) {
    return c * (Math.pow (t/d-1, 5) + 1) + b;
};

Math.easeInOutQuint = function (t, b, c, d) {
    if ((t/=d/2) < 1)
        return c/2 * Math.pow (t, 5) + b;
    return c/2 * (Math.pow (t-2, 5) + 2) + b;
};
```

This concludes the t^n easing equations. We will now look at some other mathematical operations that can produce easing curves.

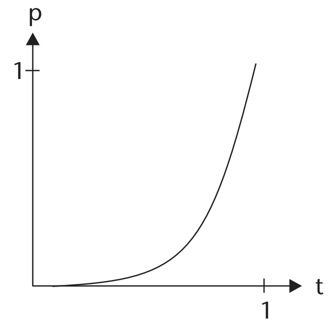

FIGURE 7-10

Graph of quintic easing

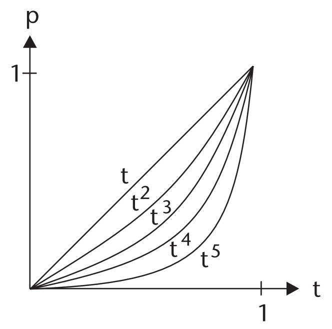

FIGURE 7-11

Graphs of t, t^2, t^3, t^4, and t^5 easing

Sinusoidal Easing

A sinusoidal equation is based on a sine or cosine function. Either one produces a *sine wave*—a periodic oscillation of a specific shape. This is the equation on which I based the easing curve:

$p(t) = \sin(t \times \pi/2)$

In the ActionScript implementation, two of the sinusoidal easing functions use cosine instead, but only to optimize the calculation. Sine and cosine functions can be transformed into each other at will. You just have to shift the curves along the time axis by one-quarter of a cycle (90 degrees or π/2 radians).

Sinusoidal easing is quite gentle, even more so than quadratic easing. Figure 7-12 shows that its path doesn't have a lot of curvature. Much of the curve resembles a straight line angled at 45 degrees, with just a bit of a curve to it.

Here are the sinusoidal easing functions:

```
Math.easeInSine = function (t, b, c, d) {
    return c * (1 - Math.cos(t/d * (Math.PI/2))) + b;
};

Math.easeOutSine = function (t, b, c, d) {
    return c * Math.sin(t/d * (Math.PI/2)) + b;
};

Math.easeInOutSine = function (t, b, c, d) {
    return c/2 * (1 - Math.cos(Math.PI*t/d)) + b;
};
```

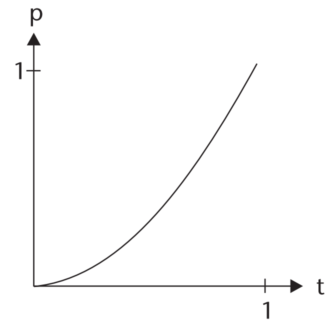

FIGURE 7-12

Graph of sinusoidal easing

Exponential Easing

I based my exponential functions on the number 2 raised to a multiple of 10:

$p(t) = 2^{10(t-1)}$

Of all the easing equations in this chapter, this one took me the longest to find. Part of the challenge is that the slope for an ease-in curve should be zero at *t*=0. An exponential curve, however, never has a slope of zero. I ended up giving the curve a very small slope that was "close enough" to zero. (If you plug *t*=0 into the preceding equation, you get 2^{-10}, which is 0.0009765625.)

Exponential easing has a lot of curvature, as shown in Figure 7-13.

The following shows the exponential easing functions:

```
Math.easeInExpo = function (t, b, c, d) {
    return c * Math.pow(2, 10 * (t/d - 1)) + b;
};

Math.easeOutExpo = function (t, b, c, d) {
    return c * (-Math.pow(2, -10 * t/d) + 1) + b;
};

Math.easeInOutExpo = function (t, b, c, d) {
    if ((t/=d/2) < 1)
        return c/2 * Math.pow(2, 10 * (t - 1)) + b;
    return c/2 * (-Math.pow(2, -10 * --t) + 2) + b;
};
```

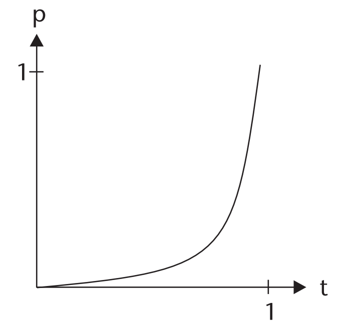

FIGURE 7-13

Graph of exponential easing

The exponential ease-out function *Math.easeOutExpo( )* produces essentially the same motion as the standard exponential slide discussed earlier. However, with my approach, you have precise control over the duration of the tween, and can jump to any point in time without running the tween from the starting point.

Circular Easing

Circular easing is based on the equation for half of a circle, which uses a square root (shown next).

$$p(t) = 1 - \sqrt{1 - t^2}$$

The curve for circular easing is simply an arc (the quarter-circle shown in Figure 7-14), but it adds a unique flavor when put into motion. Like quintic

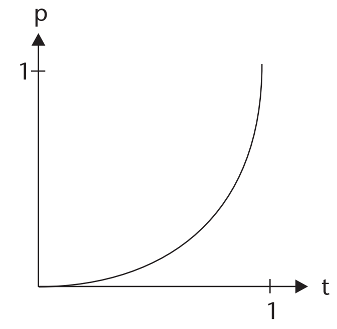

FIGURE 7-14

Graph of circular easing

and exponential easing, the acceleration for circular easing is dramatic, but somehow it seems to happen more "suddenly" than the others.

Here are the circular easing functions:

```
Math.easeInCirc = function (t, b, c, d) {
    return c * (1 - Math.sqrt(1 - (t/=d)*t)) + b;
};

Math.easeOutCirc = function (t, b, c, d) {
    return c * Math.sqrt(1 - (t=t/d-1)*t) + b;
};

Math.easeInOutCirc = function (t, b, c, d) {
    if ((t/=d/2) < 1)
        return c/2 * (1 - Math.sqrt(1 - t*t)) + b;
    return c/2 * (Math.sqrt(1 - (t-=2)*t) + 1) + b;
};
```

This concludes the tweening functions for this chapter. Like an ice cream stand, we have eight flavors of tweening: linear, quadratic, cubic, quartic, quintic, sinusoidal, exponential, and circular. Obviously, additional types of tweening are possible. All it takes is some kind of equation that takes the four essential parameters—time, beginning position, change, and duration—and calculates an appropriate position.

Introducing the Tween Class

Remember the process we went through earlier in the chapter to produce a dynamic tween? This was the resulting code:

```
// a linear tween in _x
this.begin = 100;
this.finish = 220;
this.change = this.finish - this.begin;
this.duration = 30;
this.time = 0;
this.onEnterFrame = function () {
    with (this) {
        _x = Math.linearTween (time++, begin, change, duration);
        if (time > duration) delete this.onEnterFrame;
    }
};
```

The code works well, but it's a bit unwieldy to type it all out whenever we want to create a dynamic tween. I developed a *Tween* class to simplify dynamic ActionScript movement, encapsulating the necessary elements for a tween. Using *Tween*, I can replace the preceding code with just one line:

```
x_twn = new Tween (this, "_x", Math.tweenLinear, 100, 220, 30);
```

The *Tween* instance `x_twn` handles the movement deftly, automatically starting and stopping the movement and keeping track of the time in its own internal clock. Soon, we'll examine the inner workings of the *Tween* class and learn how it manages the necessary details. But first, let's look at its superclass, *Motion*.

The Motion Class

I abstracted a lot of animation functionality into a *Motion* class. *Motion* is the superclass for several classes, one of which is *Tween* (another is the *MotionCam* class, discussed in Chapter 13). We'll look first at the *Motion* class, and then the *Tween* class.

The Motion Constructor

The constructor for the *Motion* class has a fair bit of code:

```
 _global.Motion = function (obj, prop, begin, duration, useSeconds) {
    this.setObj (obj);
    this.setProp (prop);
    this.setBegin (begin);
    this.setPosition (begin);
    this.setDuration (duration);
    this.setUseSeconds (useSeconds);
    this._listeners = [];
    this.addListener (this);
    this.start();
};
```

This isn't as complicated as it looks. Most of the code has to do with initializing properties. We'll step through it a bit at a time.

First off, we have five arguments that enter the function. Table 7-1 lists and describes the arguments for the *Motion* constructor.

Parameter	Type	Sample Value	Description
obj	object reference	mc	The object containing the affected property
prop	string	"_x"	The name of the property that will be controlled by the motion
begin	number	20	The starting position of the motion
duration	number	30	The length of time of the motion, in either frames or seconds; if the parameter is undefined or 0, the duration is infinity
useSeconds	Boolean	true	A flag asking whether to use seconds instead of frames; defaults to false

TABLE 7-1

Arguments for the *Motion* Constructor

The next three lines initialize the first three properties—obj, prop, and begin—by calling their respective setter methods:

```
this.setObj (obj);
this.setProp (prop);
this.setBegin (begin);
```

Since the motion's position should start at the specified beginning position, we set the position to begin as well:

```
this.setPosition (begin);
```

Then, the last two constructor arguments, duration and useSeconds, are passed to their respective setter methods:

```
this.setDuration (duration);
this.setUseSeconds (useSeconds);
```

Next, we declare a _listeners array property. This is required to support event broadcasting.

```
this._listeners = [];
```

A *Motion* object broadcasts custom events, like *onMotionStarted* and *onMotionChanged*. The *ASBroadcaster* object is used to enable *Motion* as an event source, as outlined in Chapter 6. The *ASBroadcaster.initialize( ) method* is invoked on the prototype after the constructor, as we'll see shortly.

The next step is to make the *Motion* instance listen to itself:

```
this.addListener (this);
```

As a result of this code, a *Motion* receives its own events. This means we can not only add other objects as listeners for *Motion* events, we can also use the *Motion*'s own event handlers as callbacks for the events.

If this seems confusing, think of a more familiar class: *TextField*. A *TextField* instance works with events in much the same way as a *Motion*. One of the *TextField* event handlers is *onChanged( )*. A text field can register listeners, all of which can respond to this event with their *onChanged( )* event handlers. However, the text field *itself* responds to the event with its own *onChanged( )* handler. Thus, we could say that a text field listens to its own events. In the *TextField* constructor, the text field adds itself as a listener, with the equivalent of this code:

```
this.addListener (this);
```

This is exactly what I have done in the *Motion* constructor. I've simply mimicked the process by which the *TextField* class handles events. I did this in order to provide maximum flexibility. Some people may want to use listeners; others will prefer simple callbacks on the *Motion* object itself. The choice is yours.

You may be wondering how I know what's inside the *TextField* constructor. The truth is, I don't know the exact code, but I have other evidence that the text field adds itself as a listener. If you create a new text field and check its `_listeners` property, you'll find it already has an object in it. The length of the `_listeners` array is 1, as the following code demonstrates:

```
tf = new TextField();
trace (tf._listeners.length); // output: 1
```

Furthermore, the first listener in the array—the object in the first index—is the text field itself, as the following code proves:

```
trace (tf._listeners[0] == tf); // true
```

The *Motion* constructor has one last task to perform. The function concludes by starting the *Motion* from the beginning:

```
this.start();
```

Now the *Motion* instance moves automatically each frame, because, as we'll see, the *start( )* method enables the object's *onEnterFrame( )* handler.

Public Methods

Adding methods is my favorite part of building a class. It's like I've created a basic shell for a gadget, and now I get to add buttons and features.

I start by defining a temporary shortcut variable, `MP`, to *Motion*'s prototype object:

```
var MP = Motion.prototype;
```

Now I can define methods in `MP`, and it's the same as defining them in `Motion.prototype`, only with shorter code.

We want *Motion* to be an event source, so we call *AsBroadcaster.initialize( )* on its prototype to empower it to that end (see Chapter 6), using the following code:

```
AsBroadcaster.initialize (MP);
```

Now let's look at the public methods that let the ActionScript user manipulate *Motion* objects.

Motion.start( )

The *start( )* method causes the *Motion* to play from the beginning:

```
MP.start = function () {
    this.rewind();
    MovieClip.addListener (this);
    this.broadcastMessage ("onMotionStarted", this);
};
```

First, the *rewind( )* method is called to send the *Motion* to its beginning point in time:

```
this.rewind();
```

Then, the object is added as a listener for *MovieClip* events:

```
MovieClip.addListener (this);
```

The *Motion* instance now receives an *onEnterFrame* event for each frame, which causes its *onEnterFrame( )* method to execute automatically. (We'll see later that *onEnterFrame( )* calls the *nextFrame( )* method.)

The last line broadcasts the *onMotionStarted* event, which invokes the *onMotionStarted( )* methods of both the *Motion* instance and its subscribed listeners. A reference to `this`, the current object, is sent in the broadcast as an argument. This allows listeners to know which object is sending the event.

Motion.stop( )

The *stop( )* method causes the *Motion* to quit moving on its own. Here's the code:

```
MP.stop = function () {
    MovieClip.removeListener (this);
    this.broadcastMessage ("onMotionStopped", this);
};
```

The first line uses the *MovieClip.removeListener( )* method to deactivate the *onEnterFrame( )* handler, stopping the frame loop. As a result, the *Motion* instance no longer moves each frame. The second line broadcasts the *onMotionStopped* event to subscribed listeners. Again, a reference to `this` is sent in the broadcast as an argument.

Motion.resume( )

The *resume( )* method causes the *Motion* object to play automatically. Unlike the *start( )* method, however, *resume( )* proceeds from the *Motion*'s current time instead of from the beginning.

```
MP.resume = function () {
    this.fixTime();
    MovieClip.addListener (this);
    this.broadcastMessage ("onMotionResumed", this);
};
```

The first line calls *fixTimer( )* to make the necessary adjustments when in timer-based mode (`useSeconds` is true). When a *Motion* is paused, the value of *getTimer( )* keeps increasing. When the *Motion* starts playing again, the *fixTime( )* method finds the new timer offset to use for the internal time.

The second line calls *MovieClip.addListener( )*, causing the *Motion* instance to receive *onEnterFrame* events. The last line broadcasts the *onMotionResumed* event to subscribed listeners, which can react accordingly with their *onMotionResumed( )* event handlers.

Motion.rewind( )

The *rewind( )* method sends the *Motion* back to its beginning point in time. Here is the code for it:

```
MP.rewind = function (t) {
    this.$time = (t == undefined) ? 1 : t;
    this.fixTime();
};
```

The first line validates the incoming `t` argument, which specifies a starting offset. If `t` isn't specified, a default value of 1 is chosen; otherwise, the `$time` property is set to the value of `t`. I'm using a dollar sign ($) in the property name to signify that it has a corresponding `time` getter/setter property, which we'll discuss later in the chapter. This is merely a personal naming convention I settled on.

Lastly, *fixTime( )*, a private method, is called to adjust the internal time-tracking offset. We'll look at *fixTime( )* later in the chapter in the "Private Methods" section.

NOTE: When *rewind( )* is called publicly, that is, by ActionScript outside the *Motion* instance, the `t` parameter isn't necessary. The parameter is used only by the *Motion.setTime( )* method internally in a special case where the *Motion* instance loops in timer-based mode.

Motion.fforward( )

The *fforward( )* method "fast-forward" the *Motion* instance to its end point. Over the course of the *Motion*, time moves between zero and the duration. Thus, the endpoint of the *Motion* is simply where the current time equals the duration. The method's code is straightforward:

```
MP.fforward = function () {
    this.setTime (this.$duration);
    this.fixTime();
};
```

The time is set to the *Motion*'s duration, and then adjusted with the private *fixTime( )* method.

Motion.nextFrame( )

The *nextFrame( )* method advances the time of the *Motion* by one frame. The code is built to accommodate the two different time modes—timer-based and frame-based:

```
MP.nextFrame = function () {
    if (this.$useSeconds) {
        this.setTime ((getTimer() - this.startTime) / 1000);
    } else {
        this.setTime (this.$time + 1);
    }
};
```

The `if` statement checks the `$useSeconds` property. If it is `true`, the *Motion* is in timer-based mode. In that case, the value of *getTimer( )* is checked against the internal offset `this.startTime`, and divided by 1000 to convert it to seconds. This value is passed to the *setTime( )* method.

If the *Motion* is in frame-based mode, the time value is simply increased by one.

NOTE: Although *nextFrame( )* is a public method, you probably won't need to call it directly in most situations, as it is called automatically by the *onEnterFrame( )* handler.

Motion.prevFrame( )

The *prevFrame( )* method sets the *Motion* back in time by one frame. Here's the code:

```
MP.prevFrame = function () {
    if (!this.$useSeconds) this.setTime (this.$time - 1);
};
```

The *prevFrame( )* method is designed to work only with frame-based motion. It's quite difficult to go backwards when you're using timer-based motion, because *getTimer( )* is always moving *forward*. Consequently, there is an `if` statement to check the `$useSeconds` property. If the property is `true`, the *Motion* instance is timer-based, and so the rest of the code is not allowed to execute. If `$useSeconds` is false, the *Motion* is frame-based, and thus, the time is decreased by 1 to go to the previous frame.

NOTE: When a *Motion* is playing automatically, the *nextFrame( )* method is called each frame by *onEnterFrame( )*. Thus, you should call the *stop( )* method before calling *prevFrame( )*. Otherwise, the calls to *prevFrame( )* and *nextFrame( )* will effectively cancel each other out in each frame.

Motion.onEnterFrame( )

One of the best features of *Motion* objects is that they can run themselves. As we saw earlier, the *start( )* method uses *MovieClip.addListener( )* to subscribe the object to *onEnterFrame* events. All we really need the *onEnterFrame( )* handler to do is advance the *Motion* to the next frame. Thus, its code is very simple:

```
MP.onEnterFrame = MP.nextFrame;
```

We just assign a reference to the *nextFrame( )* method straight across to *onEnterFrame( )*.

Motion.toString( )

The *toString( )* method provides a custom string representation of the object. I chose the three main *Motion* properties to put in the string—`prop`, `time`, and `position`:

```
MP.toString = function () {
    return "[Motion prop=" + this.prop + " t=" + this.time +
        " pos=" + this.position + "]";
};
```

This method is useful primarily for debugging purposes. When we trace a *Motion* object, its *toString( )* method is called automatically, sending a string to the Output window. In the following code example, a *Motion* is created and traced:

```
motionX = new Motion (this, "_x", 90, 20, false);
trace (motionX);
// output: [Motion prop=_x t=0 pos=90]
```

Getter/Setter Methods

The getter and setter methods of a class are public methods that provide an interface to its properties. The *Motion* class has quite a few getter and setter methods.

Motion.getPosition( )

The *getPosition( )* method is an empty placeholder function:

```
MP.getPosition = function (t) {
    // calculate and return position
};
```

Nevertheless, *getPosition( )* is the most crucial *Motion* method. Its purpose is to return the position of the *Motion* at a specified time.

If that is the case, then why is the *getPosition( )* function empty? It's what you might call an *abstract* method. It defines the interface—the external structure—of the method, but doesn't specify how the method is implemented. It is intended to be overridden in either an instance or a subclass of *Motion*. We'll see how this is done when we look at the *getPosition( )* method of the *Tween* class, which overrides *Motion.getPosition( )*.

Motion.setPosition( )

The *setPosition( )* method changes the position of the *Motion*. The following shows the code for the *setPosition( )* method:

```
MP.setPosition = function (p) {
    this.$prevPos = this.$pos;
    this.$obj[this.$prop] = this.$pos = p;
    this.broadcastMessage ("onMotionChanged", this, this.$pos);
};
```

First, the previous position is stored in a separate property `this.$prevPos`. This value can be retrieved through the *getPrevPos( )* method. Next, the incoming `p` parameter is assigned to the `this.$pos` property, and then to the controlled property referenced by `this.$obj[this.$prop]`. For instance, if the *Motion* controls the `_alpha` of a movie clip `ball`, `this.$obj[this.$prop]` translates to `ball["_alpha"]`, which is the same as `ball._alpha`. Again, I'm using dollar signs in these property names because they have corresponding getter/setter properties, defined later.

Lastly, the *onMotionChanged* event is broadcast to the *Motion*'s listeners. This causes each listener's *onMotionChanged( )* handlers to be invoked and passed two arguments: the current *Motion* object and its position. Remember that the *Motion* object listens to itself, so its own *onMotionChanged( )* handler is called here.

Motion.getPrevPos( )

At times, we may want to compare a *Motion*'s current position with the previous frame's. For example, we may need to draw a line between the two points or calculate the *Motion*'s velocity. The *getPrevPos( )* method retrieves this value for us. Here's the code:

```
MP.getPrevPos = function () {
    return this.$prevPos;
};
```

Motion.setTime( )

The purpose of the *setTime( )* method is quite simple: to change the current time of the *Motion*. The implementation, though, is a bit involved, as the method's code shows:

```
MP.setTime = function (t) {
    this.prevTime = this.$time;
    if (t > this.duration) {
        if (this.$looping) {
            this.rewind (t - this.$duration);
            this.broadcastMessage ("onMotionLooped", this);
        } else {
            this.stop();
            this.broadcastMessage ("onMotionFinished", this);
        }
    } else if (t < 0) {
        this.rewind();
    } else {
        this.$time = t;
    }
    this.update();
};
```

I've spent many hours on this method alone, rethinking and restructuring the code time and again. There are a number of conditions to check with nested `if..else` statements. If the new time is greater than the duration, the *Motion* needs to be stopped, and an *onMotionFinished* event broadcasted to listeners. The exception is that if the *Motion* is set to loop, it should rewind and broadcast a *onMotionLooped* event instead.

If the new time `t` is less than zero, the *Motion* is rewound. And if by some chance `t` is actually in the correct range and makes it through the sieve of conditionals, the *Motion*'s time is set to the value of `t`. The *update( )* method is called after all this, to bring the *Motion*'s position in line with the new time.

Motion.getTime( )

The *getTime( )* method is straightforward, returning the current time stored in the internal `$time` property:

```
MP.getTime = function () {
    return this.$time;
};
```

Motion.setBegin( ) and getBegin( )

The *setBegin( )* method defines the starting position of the *Motion*, and the *getBegin( )* method returns it. Their code is minimal:

```
MP.setBegin = function (b) {
    this.$begin = b;
};
```

```
MP.getBegin = function () {
    return this.$begin;
};
```

Motion.setDuration( ) and getDuration( )

The *setDuration( )* method defines the length of time of the *Motion*, in either frames or seconds. Here's the method's code:

```
MP.setDuration = function (d) {
    this.$duration = (d == null || d <= 0) ? Infinity : d;
};
```

I perform a simple validation here with an abbreviated `if` statement (the ternary operator). This is the logic in plain terms: If the `d` parameter is missing, negative, or zero, the *Motion* is given an infinite duration. Otherwise, the duration is set to the value of `d`.

The *getDuration( )* method returns the *Motion*'s duration in a straightforward manner:

```
MP.getDuration = function () {
    return this.$duration;
};
```

Motion.setLooping( ) and getLooping( )

By default, when a *Motion*'s time is up, it stops automatically. However, you can set a `looping` flag that causes the *Motion* to go back to the beginning instead, repeating the same movement *ad infinitum*.

The *setLooping( )* and *getLooping( )* methods provide access to the `looping` property. Their code is simple:

```
MP.setLooping = function (b) {
    this.$looping = b;
};

MP.getLooping = function () {
    return this.$looping;
};
```

Other Getter/Setter Methods

The remaining getters and setters are simple methods. They change or retrieve their respective properties without any special validation—for now, that is. The point of getter and setter methods is that they force the user to invoke a method in order to access a property. In other words, the changing of a property (or retrieval) is intercepted by a function, which can execute other code to verify or modify the data. A property may not require any validation or other associated code at first. However, you may change your class later on, and find you need to mediate access to certain properties. If you've been using getter and setter methods all along, you can just update the code in the methods and not need to change any code outside the class. All in all, although getter and setter methods may seem like overkill, they are a good practice in general for OOP projects.

The following code lists the remaining getter and setter methods:

```
MP.setObj = function (o) {
    this.$obj = o;
};
MP.getObj = function () {
    return this.$obj;
};

MP.setProp = function (p) {
    this.$prop = p;
};
MP.getProp = function () {
    return this.$prop;
};

MP.setUseSeconds = function (useSecs) {
    this.$useSeconds = useSecs;
};
MP.getUseSeconds = function () {
    return this.$useSeconds;
};
```

Private Methods

In full-fledged object-oriented programming languages like C++ and Java, private methods are accessible only to code in the same object. In ActionScript, you can't hide an object's methods from the outside environment. However, you will sometimes have methods that *should* only be used internally by the object, ideally. In these cases, it's good practice to leave comments in the code indicating that these particular methods are "private." This basically means, "you can access these methods, but you shouldn't need to, so please don't." Using private methods is like walking into a restaurant's kitchen to grab your meal: you can do it, but it's better to leave it to the waiter.

In the *Motion* class, I have two private methods: *fixTime( )* and *update( )*. They encapsulate tasks that other methods depend on.

Motion.fixTime( )

The *Motion.fixTime( )* method solves a particular problem I ran into with the *resume( )*, *rewind( )*, and *fforward( )* methods. With the *Motion* in timer-based mode (that is, when `useSeconds` is true and *Motion* is calculated with *getTimer( )*), the time needs to be "fixed" when these three methods are called. It was difficult to find the solution to this obscure obstacle, and it is likewise difficult to explain its importance. In any case, here is the code:

```
MP.fixTime = function () {
    if (this.useSeconds)
        this.startTime = getTimer() - this.$time*1000;
};
```

First, the `useSeconds` property is checked. If it is `true`, the `startTime` property is set to the current *getTimer( )* value minus the current time. Since the *Motion*'s time is stored in seconds, and *getTimer( )* and `startTime` are in milliseconds, `this.$time` is multiplied by 1000. It's really the `startTime` property that is corrected by *fixtime( )*—it is synchronized with the current time. If `startTime` is already in sync, it won't be affected by the method.

Motion.update( )

The private method *update( )* updates the *Motion*'s targeted property to reflect the position at the current time. Here's the code:

```
MP.update = function () {
    this.setPosition (this.getPosition (this.$time));
};
```

The current time is passed to the *getPosition( )* method, which calculates and returns the position. This is then passed to the *setPosition( )* method, which changes the targeted property.

Getter/Setter Properties

With the getter and setter methods defined, we might as well link them to getter/setter properties, with the following code:

```
with (MP) {
    addProperty ("obj", getObj, setObj);
    addProperty ("prop", getProp, setProp);
    addProperty ("begin", getBegin, setBegin);
    addProperty ("duration", getDuration, setDuration);
    addProperty ("useSeconds", getUseSeconds, setUseSeconds);
    addProperty ("looping", getLooping, setLooping);
    addProperty ("prevPos", getPrevPos, null);
    addProperty ("time", getTime, setTime);
    addProperty ("position",
                function() { return this.getPosition(); },
                function(p){ this.setPosition (p); } );
}
```

Because we have a consistent naming convention for the getter and setter methods, the parameters for the *addProperty( )* commands are straightforward. The name of the property comes first, as a string—"duration" for example. Then we pass a reference to the getter function, and finally, the setter function—for instance, *getDuration( )* and *setDuration( )*. In the case of the `prevPos` property, there is no setter function (it doesn't make sense to *set* this property from outside the object), so we put `null` instead of a setter function reference.

I used the `with` statement in conjuction with *Object.addProperty( )* to shorten the code and eliminate redundancy. If I don't use `with`, the code looks like this:

```
MP.addProperty ("obj", MP.getObj, MP.setObj);
MP.addProperty ("prop", MP.getProp, MP.setProp);
// etc.
```

Since everything in this situation belongs to `MP` (`Motion.prototype`), we might as well set the scope to `MP` using `with`:

```
with (MP) {
    addProperty ("obj", getObj, setObj);
    addProperty ("prop", getProp, setProp);
// etc.
}
```

The one oddity in this litany of properties is the `position` property. The code looks a bit different:

```
addProperty ("position",
            function() { return this.getPosition(); },
            function(p){ this.setPosition (p); } );
```

Because the *getPosition( )* method is designed to be overridden, we don't want to connect the `position` getter/setter property *directly* to the current *getPosition( )*. Instead, we use anonymous wrapper functions to look up the methods dynamically. Without the mediation of these wrapper functions, the `position` getter/setter property would not work properly in subclasses of *Motion*.

Finishing Off

Now that the methods have been defined, the `MP` variable is no longer needed. Thus, it is deleted this way:

```
delete MP;
```

The code for the *Motion* class is finished. We send a message to the Output window to celebrate:

```
trace (">> Motion class loaded");
```

The Tween Class

I have worked on the *Tween* and *Motion* classes off and on for almost a year, spending upwards of 80 hours on them. I originally had just a *Tween* class, but after a while, I saw that other types of motion, such as physics or recorded motion, shared much of the same infrastructure. I abstracted the overlapping functionality out of the *Tween* class and into the *Motion* superclass. We have seen how *Motion* objects keep track of time, position, movement, and other matters. Now we'll look at how the *Tween* class builds on top of this and provides a convenient, object-oriented framework for ActionScripted motion tweens.

The Tween Constructor

Once again, our exploration of a class begins with the constructor. Here's the code for the *Tween* constructor:

```
_global.Tween = function (obj, prop, func,
    begin, finish, duration, useSeconds) {
    this.superCon (obj, prop, begin, duration, useSeconds);
    this.setFunc (func);
    this.setFinish (finish);
};
```

Parameter	Type	Sample Value	Description
`obj`	object reference	`this`	The object containing the property to be tweened
`prop`	string	`"_x"`	The name of the property to tween
`func`	function reference	`Math.easeInQuad`	A tweening function used to calculate the position
`begin`	number	`100`	The starting position of the tween
`finish`	number	`220`	The finishing position of the tween
`duration`	number	`0`	The length of time of the tween, in either frames or seconds
`useSeconds`	Boolean	`true`	A flag specifying whether to use seconds instead of frames

TABLE 7-2

Arguments for the *Tween* Constructor

The constructor has seven arguments which are summarized and described in Table 7-2.

The first line of the constructor passes five of the seven arguments to the *Motion* superclass constructor:

```
this.superCon (obj, prop, begin, duration, useSeconds);
```

The remaining two arguments, `func` and `finish`, are passed to the appropriate setter methods:

```
this.setFunc (func);
this.setFinish (finish);
```

That's all there is to the *Tween* constructor; it's actually quite simple. Although seven arguments may seem like a lot, five of them are handled by the superclass, and the other two are taken care of by straightforward methods.

Immediately after the constructor, inheritance is established between the *Tween* and *Motion* classes:

```
Tween.extend (Motion);
```

This sets up the prototype chain so that *Tween*'s prototype inherits everything from *Motion*'s prototype; its methods in particular. Speaking of which, it's time to give our *Tween* class some methods of its own.

Public Methods

As usual, I define a temporary shortcut to the class prototype, in the following code:

```
var TP = Tween.prototype;
```

Because the *Tween* class inherits a number of public methods from *Motion*, it doesn't need too many more. I thought of two new public methods that would be useful, though I will probably think of more in the future.

Tween.continueTo( )

Wouldn't it be nice to have an easy way to string together several movements? For instance, you may want to tween a clip's `_alpha` to 80, then tween to 40 immediately after. The *continueTo( )* method lets you point a *Tween* to a new destination with a simple command. Here is its code:

```
TP.continueTo = function (finish, duration) {
    this.setBegin (this.getPosition());
    this.setFinish (finish);
    if (duration != undefined)
        this.setDuration (duration);
    this.start();
};
```

First, the current position of the *Tween* becomes its new starting point:

```
this.setBegin (this.getPosition());
```

Then, the incoming `finish` argument becomes the new finishing point of the *Tween*:

```
this.setFinish (finish);
```

The method also gives you the option of setting a new duration. If the duration argument is defined, the *Tween* is changed accordingly:

```
if (duration != undefined)
    this.setDuration (duration);
```

Lastly, the *Tween* is started from its new beginning point:

```
this.start();
```

The following code example shows how to use *continueTo( )* in conjunction with the *onMotionFinished( )* handler to produce a continuous string of eased movements:

```
// test Tween.continueTo()
// "ball" is an existing movie clip instance
x_twn = new Tween (ball, "_x", Math.easeInOutCirc, 20, 70, 25);
```

```
x_twn.onMotionFinished = function () {
    this.continueTo (this.position + 50);
};
```

This code causes the `ball` movie clip to start at a position (20, 0), and move to (70, 0) over 25 frames, using circular in-out easing. When `x_twn` finishes by reaching its destination, the *onMotionFinished* event is fired automatically. This invokes `x_twn`'s callback method *x_twn.onMotionFinished( )*, which causes the *Tween* to "continue to" the point 50 pixels to the right of its current position. When this new destination is reached, the *onMotionFinished* event fires again, which defines a new destination, and so on. The process repeats indefinitely, but it can be stopped either by calling the *x_twn.stop( )* method or by deleting the *x_twn.onMotionFinished( )* handler.

Tween.yoyo( )

Here's a fun method—*yoyo( )*:

```
TP.yoyo = function () {
    with (this) {
        continueTo (getBegin(), getTime());
    }
};
```

The *yoyo( )* method sends the *Motion* back towards its starting point, using the *continueTo( )* method. The current time is passed as the second parameter. This ensures that the trip backward takes the same amount of time as the trip forward. For example, if a *Tween* has a duration of 60 frames, but the *yoyo( )* method is called 25 frames into it, the *Tween* will return to its starting point in 25 frames.

Here is a code example that produces a nice down-up yo-yo motion with easing:

```
// test Tween.yoyo()
// "ball" is an existing movie clip instance
y_twn = new Tween (ball, "_y", Math.easeOutQuad, 40, 180, 20);
y_twn.onMotionFinished = function () {
    this.setFunc (Math.easeInQuad);
    this.yoyo();
    delete this.onMotionFinished;
};
```

First, a *Tween* object is created to control the `_y` property of the `ball` movie clip, taking it from 40 to 180 pixels vertically in a 20-frame time span. The *Tween.onMotionFinished( )* handler is then assigned a few actions that will execute when the *Tween* finishes:

1. The tweening function is changed from an ease-out to an ease-in. A real yo-yo starts out fast at the top and slows to a stop at the bottom—an ease-out. It then speeds up as it rises, stopping suddenly at the top—an ease-in.
2. The *Tween.yoyo( )* method is called to swap the start and end points of the *Tween*.
3. The *onMotionFinished( )* handler is deleted because we want it to only execute once. If you eliminate this step, the ball will oscillate indefinitely.

Getter/Setter Methods

The *Tween* class adds three getter/setter properties: `func`, `change`, and `finish`. We first define the getter/setter methods for these, then connect them to the properties with *Object.addProperty( )* (illustrated later in the "Getter/Setter Properties" section of this chapter). But first, we take care of the all-important *getPosition( )* method.

Tween.getPosition( )

The *getPosition( )* method returns the position of the *Tween* at a certain time, and overrides the superclass method *Motion.getPosition( )* (which is an empty function by default). Here's the code:

```
TP.getPosition = function (t) {
    if (t == undefined) t = this.$time;
    with (this) return $func (t,
                              $begin,
                              $change,
                              $duration);
};
```

In the first line, if a specific time is not given through the `t` argument, the current time is chosen. Next, the properties for the time, beginning position, change in position, and duration are passed to the easing function. The resulting value is returned from the method.

Tween.setFunc( ) and getFunc( )

The *setFunc( )* and *getFunc( )* methods govern the `func` getter/setter property, represented internally by the `$func` property. My tweening functions are ideal candidates for `func`, although you can define your own, as long as they conform to the same basic structure and arguments (`t`, `b`, `c`, and `d`).

Here's the code for the methods:

```
TP.setFunc = function (f) {
    this.$func = f;
};

TP.getFunc = function () {
    return this.$func;
};
```

Tween.setChange( ) and getChange( )

The *setChange( )* and *getChange( )* methods mediate access to the `change` property, which stores the change in position from the beginning of the *Tween* to the end.

NOTE: In mathematical function terminology, you could call the change of a tween its *range*, and its duration the tween's *domain*.

The code for these methods is straightforward:

```
TP.setChange = function (c) {
    this.$change = c;
};

TP.getChange = function () {
    return this.$change;
};
```

Tween.setFinish( ) and getFinish( )

The `finish` getter/setter property governs the finishing position of the *Tween*. It is implemented a little differently than previous properties. The value of `finish` *per se* is not stored as its own property. Rather, it is defined in terms of the `change` property. The two are interdependent: you can't modify `finish` without modifying `change`. Thus, I only maintain one internal property—for `change`, calculating `finish` as needed. I chose `change` because my tweening functions use a change argument (rather than a finish argument).

You can see in the following code how a specified value for `finish` is converted into the equivalent `change` value:

```
TP.setFinish = function (f) {
    this.$change = f - this.$begin;
};
```

The beginning position is subtracted from the finishing point to yield the change in position.

Likewise, the *getFinish( )* method calculates the `finish` value on the fly by adding `change` to `begin`:

```
TP.getFinish = function () {
    return this.$begin + this.$change;
};
```

Getter/Setter Properties

The *Tween* class inherits the nine getter/setter properties of the *Motion* superclass. It also adds three more getter/setter properties—`func`, `change`, and `finish`—with the following code:

```
with (TP) {
    addProperty ("func", getFunc, setFunc);
    addProperty ("change", getChange, setChange);
    addProperty ("finish", getFinish, setFinish);
}
```

Some people will prefer getter/setter properties; others will choose to call the methods directly. It's a matter of personal preference. Personally, I use the methods most of the time; it's slightly faster than calling the getter/setter property, which causes the run-time interpreter to look up the method for you.

Finishing Off

With the constructor, methods and properties defined, our *Tween* class is complete. It's time to clean up by deleting the shortcut variable `TP` and sending a message to the Output window, with the last bit of code:

```
delete TP;
trace (">> Tween class loaded");
```

Conclusion

In this chapter, we have looked at dynamic motion from a particular angle—where there is one definite position for a given time. We dissected the concept of easing and looked at several examples of easing curves and functions. These concepts are encapsulated in a practical manner in the *Motion* and *Tween* classes. In the next chapter, we'll look at motion produced by a different process—physics animation.