

# Object-Oriented ActionScript for Flash 8

Peter Elst and Todd Yard  
with Sas Jacobs and William Drol



# Object-Oriented ActionScript for Flash 8

Copyright © 2006 by Peter Elst, Todd Yard, Sas Jacobs, and William Drol

All rights reserved. No part of this work may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording, or by any information storage or retrieval system, without the prior written permission of the copyright owner and the publisher.

ISBN-13 (pbk): 978-1-59059-619-7

ISBN-10 (pbk): 1-59059-619-6

Printed and bound in the United States of America 9 8 7 6 5 4 3 2 1

Trademarked names may appear in this book. Rather than use a trademark symbol with every occurrence of a trademarked name, we use the names only in an editorial fashion and to the benefit of the trademark owner, with no intention of infringement of the trademark.

Distributed to the book trade worldwide by Springer-Verlag New York, Inc., 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax 201-348-4505, e-mail orders-ny@springer-sbm.com, or visit [www.springeronline.com](http://www.springeronline.com).

For information on translations, please contact Apress directly at 2560 Ninth Street, Suite 219, Berkeley, CA 94710. Phone 510-549-5930, fax 510-549-5939, e-mail [info@apress.com](mailto:info@apress.com), or visit [www.apress.com](http://www.apress.com).

The information in this book is distributed on an “as is” basis, without warranty. Although every precaution has been taken in the preparation of this work, neither the author(s) nor Apress shall have any liability to any person or entity with respect to any loss or damage caused or alleged to be caused directly or indirectly by the information contained in this work.

The source code for this book is freely available to readers at [www.friendsofed.com](http://www.friendsofed.com) in the Downloads section.

## Credits

**Lead Editor**      **Assistant Production Director**  
Chris Mills      Kari Brooks-Copony

**Technical Reviewers**      **Production Editor**  
Jared Tarbell,      Katie Stence  
Stephen Downs

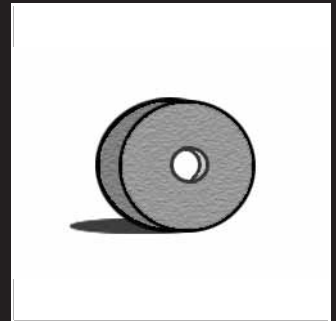
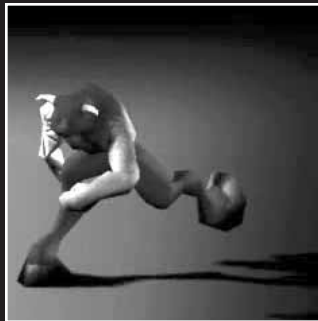
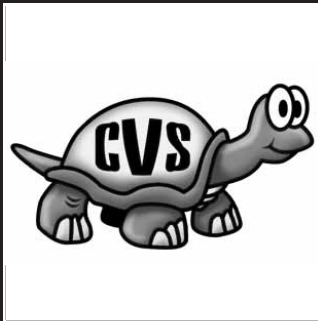
**Editorial Board**      **Compositor**  
Steve Anglin, Dan Appleman,      Dina Quan  
Ewan Buckingham, Gary Cornell,      **Proofreader**  
Jason Gilmore, Jonathan Hassell,      April Eddy  
James Huddleston, Chris Mills,      **Indexer**  
Matthew Moodie, Dominic Shakeshaft,      Michael Brinkman  
Jim Sumser, Matt Wade

**Project Manager**      **Artist**  
Sofia Marchant      April Milne

**Copy Edit Manager**      **Interior and Cover Designer**  
Nicole LeClerc      Kurt Krames

**Copy Editor**      **Manufacturing Director**  
Ami Knox      Tom Debolski

## 9 INHERITANCE



In this chapter, I'll walk you through an example to demonstrate inheritance in ActionScript 2.0. Whenever you write a new class to extend or enhance an existing class (without actually altering the existing class), you're using inheritance. Inheritance can introduce new capabilities without the fear of breaking existing applications.

## About class hierarchy

Inheritance groups two or more classes together into a hierarchy (much like the folder structure on your computer). The first class in the hierarchy is the *base class* (like a top-level folder). The next class is a *subclass* (like a subfolder). Each subsequent class inherits from the previous one, so every subclass has a definite parent.

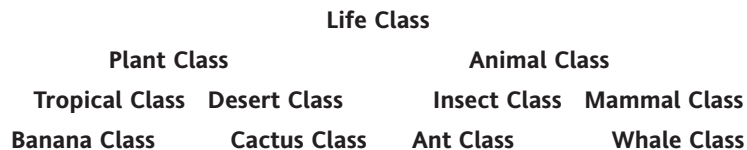
Look at the following class hierarchy:

**Animal Class** (this is the base class, it has no parent, its child is Mammal)

**Mammal Class** (this is a subclass, its parent is Animal, its child is Whale)

**Whale Class** (this is a subclass, its parent is Mammal, it has no children)

Classes range from general to specific. The base class is the most general; subclasses are more specific (for example, a *whale* is a specific *mammal*). Class hierarchies can be simple or complex. Here's another example:



Life is the base class (the most general); everything else is a subclass (more specific). The Ant class inherits from the Insect class. The Whale class inherits from the Mammal class. The Ant and Whale classes inherit (indirectly) from the Animal class, but they don't even know it! The Ant class only communicates with the Insect class, and the Whale class only communicates with the Mammal class.

*This is natural inherited behavior in OOP—a class may only communicate with its parent (not its grandparent). ActionScript 2.0 provides a special keyword, `super`, to establish child-to-parent communications.*

## A quick inheritance test

Let's do a quick inheritance test. Using the Mover class we built in the previous chapter, let's now extend it to include some bounce behavior. Just to remind ourselves, this is the code for the Mover class:

```

class Mover extends MovieClip {
    var targetMC:MovieClip;

    function Mover(targetMC:MovieClip) {
        this.targetMC = targetMC;
    }
    function updatePosition() {
        this._x++;
        this._y++;
    }
    function startMoving() {
        this.targetMC.onEnterFrame = this.updatePosition;
    }
    function stopMoving() {
        this.targetMC.onEnterFrame = null;
    }
}

```

Before we move any further, it is important to add a little functionality that allows our Mover class to do something more than just move at the same speed both horizontally and vertically. To do this, we'll add three properties: `xVel` and `yVel`, which store the velocity at which our `targetMC` moves in any given direction; and `objectRef`, which acts as a reference to our class.

```

var xVel:Number;
var yVel:Number;
var objectRef:Object;

```

Next, we add two additional parameters to the constructor method:

```

function Mover(targetMC:MovieClip, xVel:Number, yVel:Number) {
    this.targetMC = targetMC;
    this.objectRef = this;
    this.xVel = xVel;
    this.yVel = yVel;
}

```

Finally, the `updatePosition` method needs to be tweaked to read as follows:

```

function updatePosition() {
    this._x += this.objectRef.xVel;
    this._y += this.objectRef.yVel;
}

```

The reason why we're using this `objectRef` property is because the `updatePosition` method is called from the scope of our `targetMC`, when we use `this.xVel` in that function, it looks for a property called `xVel` inside the movie clip. Using `this.objectRef.xVel`, things are different; `this.objectRef` is a reference to our class, and as a result it looks for the property `xVel` in the Mover class scope, which is exactly what we need.

*There are more convenient ways to handle scope issues such as this. I'll talk about those in Chapter 15 when discussing handling events using the `EventDispatcher` class.*

Having made these tweaks, we can now have our `Mover` class move any movie clip with any given horizontal and vertical velocity. Pretty neat! The full code of the `Mover` class now looks as follows:

```
class Mover extends MovieClip {

    var objectRef:Object;
    var targetMC:MovieClip;
    var xVel:Number;
    var yVel:Number;

    function Mover(targetMC:MovieClip, xVel:Number, yVel:Number) {
        this.targetMC = targetMC;
        this.targetMC.objectRef = this;
        this.xVel = xVel;
        this.yVel = yVel;
    }
    function updatePosition() {
        this._x += this.objectRef.xVel;
        this._y += this.objectRef.yVel;
    }
    function startMoving() {
        this.targetMC.onEnterFrame = this.updatePosition;
    }
    function stopMoving() {
        this.targetMC.onEnterFrame = null;
    }
}
```

When you want to create an instance of this new `Mover` class, you would use the following code:

```
var myMover:Mover = new Mover(circle, 2, 3);
```

This code moves our `circle` movie clip at a velocity of 2 pixels horizontally and 3 pixels vertically once every frame.

Now, the next step is writing our `Bouncer` class, which extends (or inherits) the new `Mover` class code. The code for this is listed here:

**Bouncer.as**

```
class Bouncer extends Mover {
    function Bouncer(targetMC:MovieClip) {
        super(targetMC,xVel,yVel);
    }
}
```

The preceding code needs to be saved in a file called `Bouncer.as` in the same folder as `Mover.as`.

What you'll need to do next is get a copy of `Mover.fla` and save it as `Inheritance.fla` in the folder where `Bouncer.as` and `Mover.as` are located. Open up `Inheritance.fla` and on Frame 1 of the `ActionScript` layer change the code to

```
var myBouncer:Bouncer = new Bouncer(circle, 2, 3);
myBouncer.startMoving();
```

Feel free to delete the rectangle movie clip from the stage, as we'll not be using this just now. Test the movie (select `Control > Test Movie`) and you'll see that the circle on stage is now moving, just as was the case with the `Mover` class. Nothing spectacular, you say? Well, remember, you're now instantiating the `Bouncer` class, which shows you that it has inherited the functionality of the `Mover` class by its ability to use the `startMoving` method and the `targetMC` property. Inheritance in action, what an awe-inspiring sight!

## About inheritance syntax

Let's look at what syntax we used to initiate class inheritance in `ActionScript 2.0`. First of all, you'll need to have the `extends` keyword when defining the class.

```
class Bouncer extends Mover {
    ...
}
```

After the `extends` keyword, you define what class it inherits from, in this case `Mover`. Important to note is that a class can extend only one class at a time, and this class needs to be specified by its full package name.

As soon as this `extends` syntax is added to the class statement, all methods and properties of that inherited class are available in the current class. This brings up an important point: what if we define a method or property with the same name in the `Bouncer` class? It's easy enough to give this a try—add the following code to the `Bouncer` class:

```
function startMoving() {
    trace("startMoving function called in Bouncer class");
}
```

If we use Test Movie now, we get the following line in the Output panel, but our circle movie clip does not move at all:

startMoving function called in Bouncer class

That's not good at all. We also want to have access to the `startMoving` method that was defined in the `Mover` class. This is where the `super` keyword comes in; it specifically tells a class to look for a method or property in its *parent* class, our *superclass*. Using this keyword, we can tweak the `startMoving` method in the `Bouncer` class to read as follows:

```
function startMoving() {
    trace("startMoving function called in Bouncer class");
    super.startMoving();
}
```

Testing our movie now gives us a much better response: we get both the trace statement in the Output panel and our circle is moving on the stage. It's getting more and more interesting by the minute—we've just extended a method and in the process used an important concept in OOP called **polymorphism** (more about this in the next chapter).

When you look at the code of our `Bouncer` class, you'll notice that the constructor also uses the `super` keyword but actually calls it as you would with any other method. It passes the `targetMC`, `xVel`, and `yVel` parameters we got in the `Bouncer` class down to the constructor of the `Mover` class.

In other words, a call to `super()` calls the superclass constructor; in this case, it was needed because `Mover` required a `targetMC` property to be passed to its constructor.

## The Bouncer class

Using inheritance, you can safely extend existing classes with new or alternative behavior without breaking existing applications. Existing applications continue to work because they do not interact with (or even know about) the newer classes (they use the existing classes as always).

Currently, the `Bouncer` class behaves just like the `Mover` class. Next, we'll add new behavior to the `Bouncer` class. This new behavior will consist of a `bounceAtBorder` method that bounces our movie clip when it hits the end of the stage.

```
function bounceAtBorder() {
    if (this._x > Stage.width-(this._width/2)) {
        trace("Bounce at right edge");
        this._x = Stage.width-(this._width/2);
        this.objectRef.xVel *= -1;
    }
    if (this._y > Stage.height-(this._height/2)) {
        trace("Bounce at bottom edge");
        this._y = Stage.height-(this._height/2);
        this.objectRef.yVel *= -1;
    }
}
```

```

    }
    if (this._x < this._width/2) {
        trace("Bounce at left edge");
        this._x = this._width/2;
        this.objectRef.xVel *= -1;
    }
    if (this._y < this._height/2) {
        trace("Bounce at top edge");
        this._y = this._height/2;
        this.objectRef.yVel *= -1;
    }
}

```

When you look at the `bounceAtBorder` method, you'll find that it has four `if` statements, one for each edge of the screen. Each of the `if` statements checks the `_x` or `_y` position of `targetMC` against the minimum or maximum width and height. To do this, we use the built-in `Stage.width` and `Stage.height` methods that return the available width and height of the stage. Because the registration point of our movie clip is in the center, we need to accommodate the maximum and minimum width with the `targetMC`'s `_width` and `_height` properties. Take a look at the following table to see how these minimum and maximum positions are calculated:

---

Right edge	Width of the stage minus half the width of the <code>targetMC</code> movie clip
Bottom edge	Height of the stage minus half the height of the <code>targetMC</code> movie clip
Left edge	Half the width of the <code>targetMC</code> movie clip
Top edge	Half the height of the <code>targetMC</code> movie clip

---

9

The only thing that's left for us to do now is tweak the `startMoving` method; instead of just calling `updatePosition` in the `onEnterFrame` event, we want to call both `updatePosition` and `bounceAtBorder`. The revised code for `startMoving` in the `Bouncer` class is as follows:

```

function startMoving() {
    this.targetMC.updatePosition = this.updatePosition;
    this.targetMC.bounceAtBorder = this.bounceAtBorder;
    this.targetMC.onEnterFrame = function() {
        this.updatePosition();
        this.bounceAtBorder();
    };
}

```

The preceding code makes local references in the `targetMC` movie clip to the `updatePosition` method (inherited from the `Mover` class) and `bounceAtBorder` method (from the `Bouncer` class) in our class. After that, we can call the `updatePosition` and `bounceAtBorder` methods as local in the `targetMC` `onEnterFrame` handler using `this.updatePosition` and `this.bounceAtBorder`.

## OBJECT-ORIENTED ACTIONSCRIPT FOR FLASH 8

The complete code for our Bouncer class now looks as follows:

```
class Bouncer extends Mover {

    function Bouncer(targetMC:MovieClip, xVel:Number, yVel:Number) {
        super(targetMC, xVel, yVel);
    }
    function startMoving() {
        this.targetMC.updatePosition = this.updatePosition;
        this.targetMC.bounceAtBorder = this.bounceAtBorder;
        this.targetMC.onEnterFrame = function() {
            this.updatePosition();
            this.bounceAtBorder();
        };
    }
    function bounceAtBorder() {
        if (this._x>Stage.width-(this._width/2)) {
            trace("Bounce at right edge");
            this._x = Stage.width-(this._width/2);
            this.objectRef.xVel *= -1;
        }
        if (this._y>Stage.height-(this._height/2)) {
            trace("Bounce at bottom edge");
            this._y = Stage.height-(this._height/2);
            this.objectRef.yVel *= -1;
        }
        if (this._x<this._width/2) {
            trace("Bounce at left edge");
            this._x = this._width/2;
            this.objectRef.xVel *= -1;
        }
        if (this._y<this._height/2) {
            trace("Bounce at top edge");
            this._y = this._height/2;
            this.objectRef.yVel *= -1;
        }
    }
}
```

You can now save `Bouncer.as` and take a look at `Inheritance.fla`. Make sure Frame 1 of the ActionScript layer reads as follows:

```
var myBouncer:Bouncer = new Bouncer(circle,2,3);
myBouncer.startMoving();
```

Save `Inheritance.fla` and run Test Movie, and you should now see the circle movie clip moving across the screen with a horizontal velocity of 2 pixels per frame and 3 pixels per frame vertically. As soon as the movie clip hits an edge, a trace statement is executed showing that in the Output panel, and it bounces off in the opposite direction.

Great work—we’ve just seen inheritance in action and built our very own Bouncer class by extending the Mover class!

## The Gravity class

Let’s create a Gravity class to help the Ball move in a natural manner. You’ll use inheritance again, but this time you’ll extend the Bouncer class.

Create a new file called `Gravity.as` and save it in the same folder as the Bouncer and Mover classes. We’ll start off with the basic code as follows:

### Gravity.as

```
class Gravity extends Bouncer {
    var strength: Number;
    function Gravity(targetMC:MovieClip, xVel:Number, yVel:Number,
strength) {
        super(targetMC, xVel, yVel);
        this.strength = strength || 1;
    }
}
```

The Gravity class constructor takes the three parameters we know are needed for the Bouncer class as well as a strength parameter that defines the strength of the gravitational pull. We pass the first three parameters to the superclass, and the third is assigned to our class scope using the following syntax:

```
this.strength = strength || 1;
```

You might be wondering what’s happening there. This is just shorthand syntax for writing the following statement:

```
if(strength == undefined) {
    this.strength = 1;
} else {
    this.strength = strength;
}
```

It essentially sets a default value of 1 in case no strength parameter gets passed to the Gravity class. The next thing we’ll do is add the `startMoving` method to the Gravity class, which will now include two new methods, `applyGravity` and `applyFriction` (we’ll be writing those in a minute), in the `targetMC.onEnterFrame` handler. You can see how that `startMoving` method now looks:

```
function startMoving() {
    this.targetMC.updatePosition = this.updatePosition;
    this.targetMC.bounceAtBorder = this.bounceAtBorder;
    this.targetMC.applyGravity = this.applyGravity;
```

## OBJECT-ORIENTED ACTIONSCRIPT FOR FLASH 8

```
        this.targetMC.applyFriction = this.applyFriction;
        this.targetMC.onEnterFrame = function() {
            this.updatePosition();
            this.bounceAtBorder();
            this.applyGravity();
            this.applyFriction();
        };
    }
}
```

Just like we did in the Bouncer class, we're making a local reference to the `applyGravity` and `applyFriction` methods in `targetMC` and calling those in `onEnterFrame`.

Now that we've got that sorted, let's focus on these two new methods. The first one we'll look at is `applyGravity`:

```
function applyGravity() {
    this.objectRef.yVel += this.objectRef.strength;
}
```

Doesn't that look easy? That code adds more pull to the vertical movement of our `targetMC` movie clip. You'll remember that `objectRef` property from the `Mover` class we updated; it's nothing more than a reference to the class scope. As you'll see, the `applyFriction` method will not be much more difficult:

```
function applyFriction() {
    this.objectRef.xVel *= 0.98;
    this.objectRef.yVel *= 0.98;
}
```

In this method, we're adding friction to both the horizontal and vertical velocity by decreasing the velocity slightly every frame. We're using 0.98 here because it gives us the best effect. You could add this friction factor as an additional parameter for your `Gravity` class. Whatever the number is you'll use for this, it should be below 1; otherwise, it would in fact increase velocity.

*It's nice to think about the friction coefficient as a percentage of the previous velocity. For example, with a coefficient of 0.98, our Mover moves only 98% the speed it did the frame before. Eventually, friction will slow the movement to nothing, just as in the physical world.*

Save the `Gravity` class and turn your attention to `Inheritance.fla`. Make sure the code in Frame 1 of the `ActionScript` layer looks as follows:

```
var myGravity:Gravity = new Gravity(circle, 2, 3, 5);
myGravity.startMoving();
```

If you save the document and run Test Movie, you'll see the circle movie clip, moving with a horizontal velocity of 2 pixels and a vertical velocity of 3 pixels per frame, bounce off the edges of the stage and slow down as if gravity were pulling it to the ground, and eventually the `targetMC` will rest on the bottom of the stage. In this case, the amount of gravity we applied is 5 pixels per frame.

One thing you will notice is that even when the movie clip has come to a standstill at the bottom of the stage, the `onEnterFrame` event continuously keeps firing the “Bounce at bottom edge” trace statement to the Output panel. To optimize our code and stop this from happening, we'll add a method that monitors the `yVel` property, and if that is zero, the `onEnterFrame` event on our `targetMC` movie clip gets cleared. This method, which we'll call `checkForStop`, needs the following code:

```
function checkForStop() {
    if (this._y == Stage.height-(this._height/2)) {
        if (Math.round(this._x) == this.objectRef.lastPosX
        && Math.round(this._y) == this.objectRef.lastPosY) {
            this.onEnterFrame = null;
        }
    }
    this.objectRef.lastPosX = Math.round(this._x);
    this.objectRef.lastPosY = Math.round(this._y);
}
```

That's quite a bit of code just to check whether our `targetMC` is still moving or not, but you'll see what it does in a minute. The first `if` statement in the `checkForStop` method checks whether the movie clip is on the bottom edge of the stage. If that is the case, it moves on to an `if` statement that checks the current `_x` and `_y` position of the `targetMC` against the last position. To do this, two new class properties are introduced: `lastPosX` and `lastPosY`, which hold the latest `_x` and `_y` position of `targetMC`. If the current `_x` and `_y` position is the same as the last `_x` and `_y` position, and the `targetMC` is on the bottom edge, we can be sure that our movie clip has stopped moving. In that case, we set the `onEnterFrame` event for the `targetMC` to `null`, essentially clearing it out.

That's it—we've just completed the Gravity class! Be sure to save `Gravity.as` and run Test Movie on `Inheritance.fla`. You'll see that the previous problem with the `onEnterFrame` event not stopping after the movie clip finished moving is now solved. The following is the full code of the Gravity class:

### Gravity.as

```
class Gravity extends Bouncer {
    var strength:Number;
    var lastPosX:Number;
    var lastPosY:Number;
```

## OBJECT-ORIENTED ACTIONSCRIPT FOR FLASH 8

```
function Gravity(targetMC:MovieClip, xVel:Number, yVel:Number,
    strength) {
    super(targetMC, xVel, yVel);
    this.strength = strength;
}
function startMoving() {
    this.targetMC.updatePosition = this.updatePosition;
    this.targetMC.bounceAtBorder = this.bounceAtBorder;
    this.targetMC.applyGravity = this.applyGravity;
    this.targetMC.applyFriction = this.applyFriction;
    this.targetMC.checkForStop = this.checkForStop;
    this.targetMC.onEnterFrame = function() {
        this.updatePosition();
        this.bounceAtBorder();
        this.applyGravity();
        this.applyFriction();
        this.checkForStop();
    };
}
function applyGravity() {
    this.objectRef.yVel += this.objectRef.strength;
}
function applyFriction() {
    this.objectRef.xVel *= 0.98;
    this.objectRef.yVel *= 0.98;
}
function checkForStop() {
    if (this._y == Stage.height-(this._height/2)) {
        if (Math.round(this._x) == this.objectRef.lastPosX
            && Math.round(this._y) == this.objectRef.lastPosY) {
            this.onEnterFrame = null;
        }
    }
    this.objectRef.lastPosX = Math.round(this._x);
    this.objectRef.lastPosY = Math.round(this._y);
}
}
```

*You might remember that we hard-coded the value of 0.98 in the applyFriction method of the Gravity class. As an exercise, try updating the code so people can specify the amount of friction when instantiating the class. Be sure to have a default value for friction. (Hint: study the strength property in the constructor of the Gravity class.)*

## Inheritance summary

Inheritance extends existing classes with new and alternative behaviors without breaking existing applications. Existing applications continue to work because they do not interact with (or even know about) the newer classes. Inheritance adds capability without breaking compatibility. Look at this sample code:

Animal.as

```
class Animal {
    function Animal() {
    }
    function speak = function(sound) {
        trace(sound);
    }
}
```

Cat.as

```
class Cat extends Animal {
    function Cat() {
    }
    function cryBaby = function() {
        for(i = 0; i < 100; i++) {
            super.speak("Meow! Meow! Meow!");
        }
    }
}
```

```
var suki:Cat = new Cat();
suki.cryBaby();
```

With just one statement

```
class Cat extends Animal {
    ...
}
```

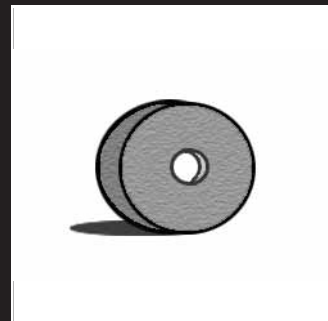
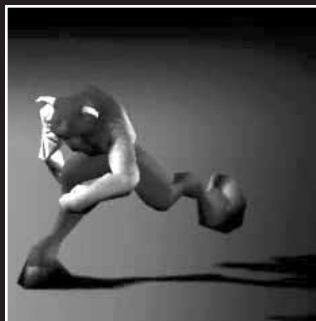
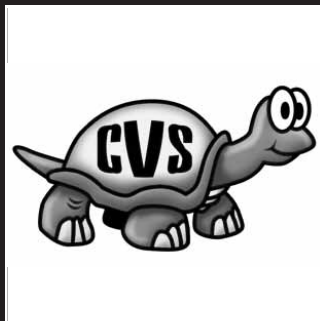
the Cat class inherits from the Animal class. The Cat class has all the capabilities and characteristics of the Animal class, plus its own unique capabilities. The `super` keyword enables child classes (such as Cat) to talk to parent classes (such as Animal). If you try the previous code, you'll find out that you can use ActionScript 2.0 to code a very noisy cat.

## What's next?

This chapter introduced the major features and benefits of class inheritance. Next, we'll look at the final building block of OOP: polymorphism.

(EXCERPT FROM)

# 17 OOP ANIMATION AND EFFECTS



The original purpose of Flash was animation. Using the timelines, animators either drew frame-by-frame graphics to simulate motion or used Flash's tweening capabilities to create animation between keyframes. With the addition of ActionScript, animation did not go out the window. In fact, ActionScript proved to be just another tool, like the timeline, that developers could use to create animation. By altering visual properties of movie clips through code, fluid and complex animation could be created without timelines. But how exactly can we apply OOP to this process of animation in order to make it easier and more manageable?

The first thing to consider is that in the case of animation classes, most likely these classes won't themselves be visible, but will instead handle the animation of other visual objects. For instance, a movie clip class that can tween its position would be extremely useful, but more useful would be a class outside of that movie clip class that handles the tweening animation. This tweening class could then be applied to other objects as well, not just the single movie clip class. If we think of these animation classes as handling animation as opposed to visually animating themselves, we can focus in on the useful pieces that would comprise such classes.

Events being fired when an animation begins, is occurring, or ends would be the most important feature, and this can be handled using the Broadcaster class created in Chapter 13, with perhaps a little enhancement. This could serve as the base class for all of our animation classes. Generally, if we have a class that can broadcast when it's animating and we can pass that class a reference to a clip that we wish to animate, we have the basis for any animation class. How the class handles the animation of the clip through some looping mechanism (perhaps an interval using our centralized IntervalManager) would be up to each animation class individually.

In this chapter, we'll explore a number of different classes to control the animation of movie clips in an object-oriented manner, playing a bit with the functionality of filters and bitmap manipulation new to Flash 8. Making a break from the process of previous chapters, we'll dive right into the code without spending too much time in the planning stages. Sometimes, especially when creating code for animation and effects, a lot of experimentation happens to get what is wanted, and creating too rigid of a box to work in makes it difficult to break out of. After the structure and discipline applied in the previous chapters (and now that you are comfortable with when and how to create UML diagrams to help with the planning), let's give ourselves a little freedom to have some fun in this chapter.

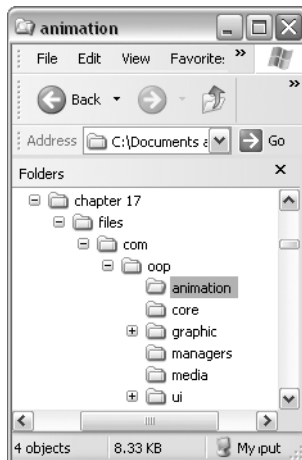
## Preparing for animation

The base class of the animators we create will use Broadcaster (from the classes we created in Chapters 13, 15, and 16) in order to dispatch events to listeners detailing when an animation begins, is occurring, and ends. As we won't know any additional information about the type of animation in this superclass, there won't be too much code we need to add.

## Animator

The first thing we need to do is create a new package for our animation classes; do this in the class directory that you first created in Chapter 13 (this could be in the Flash configuration directory, in the same directory of whatever FLA you are currently working on, or in some directory you have specified in the Flash IDE). Add a new folder/directory named `animation` in the `com/ooop` directory, as shown in Figure 17-1. We'll save our animation classes here.

17



**Figure 17-1.**  
The path to the new animation package

Create a new ActionScript file in Flash and add these lines:

```
class com.ooop.animation.Animator extends com.ooop.core.Broadcaster {
    function Animator() {}

    // PUBLIC
    // _____

    public function dispatchEvent(evt:String, params:Object):Void {
        if (params == undefined) var params:Object = {};
        params.animator = this;
        super.dispatchEvent(evt, params);
    }
}
```

Save this completed file as `Animator.as` into the `com/ooop/animation` directory. As I said, there's not much to add to this class on top of the `Broadcaster` functionality. You can see from the code that it simply inherits from `Broadcaster` and then overrides the `dispatchEvent()`. In `Animator's dispatchEvent()`, it adds a new property to the `params` object, `animator`, which stores a reference to the `Animator` instance itself (note that the

first line first checks to see whether a params object was sent to the method and, if not, creates one). The superclass, Broadcaster, then gets its `dispatchEvent()` method invoked with the altered params object passed in.

It actually makes a lot of sense to add similar functionality to Broadcaster itself, an oversight from its initial coding. In fact, if Broadcaster itself added its reference to all events it dispatched, it would remove the necessity for Animator. However, we'll leave the Animator class for two reasons. First, any animation classes we create can now inherit from Animator instead of Broadcaster, which not only makes more sense in the code, but also gives us the freedom down the line to add additional functionality to all animation classes (and not all broadcasting classes). Second, the property we have added to the params object, `animator`, will be descriptive and helpful, as we are coding listeners to animator objects. The generic term that we would have to add to Broadcaster (“target” or “broadcaster”, perhaps) would not read as clearly in the code.

## Tweening properties and values

With the base class coded, it's time to get into some actual animation. The first thing we'll look at is how to tween simple movie clip properties in order to create movement. We can contain this functionality in a single class, named Tweener, which will handle taking an object and changing a specific value for that object over the course of time.

### Tweener

Create a new ActionScript file and save it as `Tweener.as` into the `com/ooop/animation` directory. Let's begin with its basic blueprint.

```
import com.ooop.animation.Animator;
import com.ooop.managers.IntervalManager;

class com.ooop.animation.Tweener extends Animator {

    function Tweener() {}

    // PRIVATE
    // _____

    private function runTween():Void {
    }

    // PUBLIC
    // _____

    public function haltTween() {
    }
}
```

```

    public function tween():Void {
    }

    public function tweenTo():Void {
    }
}

```

17

At the top of the class, we import the two classes we know will be used, Animator (the superclass) and IntervalManager (to handle the interval calls). The four methods defined are the four that are immediately apparent. `tween()` and `tweenTo()` would initiate animation, `haltTween()` would end an animation, and `runTween()` would handle altering values to create that animation. The next step would be to decide what you would need to pass in order to start an animation.

```

    public function tween(
        clip:Object,
        prop:String,
        startValue:Number,
        destValue:Number,
        time:Number
    ):Void {
    }

```

To create a tween for an object's property, we would need to know the object, the property to tween, the values to tween, and the time in which that animation should occur. With these values passed in, we can start the process of animation by creating an interval that will be called to change the property over the course of time.

```

    public function tween(
        clip:Object,
        prop:String,
        startValue:Number,
        destValue:Number,
        time:Number
    ):Void {
        clip[prop] = startValue;
        var intName:String = clip._name + "_" + prop;
        if (__intervals[intName] == undefined) __intervals[intName] = {};
        var intObj:Object = __intervals[intName];
        intObj.count = 1;
        intObj.totalInts = Math.floor(time/Tweener.intervalTime);
        intObj.startProp = startValue;
        intObj.endProp = destValue;
        intObj.changeProp = intObj.endProp - intObj.startProp;
        intObj.clip = clip;
        intObj.prop = prop;
        intObj.tweener = this;
    }

```

```

        IntervalManager.setInterval(intObj, "interval",
➤ this, "runTween", Tweener.intervalTime, intObj);
        runTween(intObj);
        dispatchEvent("startTween", {clip:clip, prop:prop});
    }

```

The first thing that is done in this method is the clip passed in immediately gets its property set to the start value passed in. Next, we need a way for Tweener to store the interval that will be passed to the IntervalManager. This is necessary since it's possible that Tweener might handle animating multiple properties or multiple objects, so an individual and unique object will need to be passed to the IntervalManager for each interval that needs to be called. This interval object will be stored in a new property, `__intervals`, which we'll declare in a moment.

The name of the interval object will be determined by the name of the clip passed in, plus an underscore, plus the name of the property to be tweened. Thus, if a movie clip named "ball" was passed in and the property to tween was its `_rotation`, then the interval object would be stored in Tweener's `__intervals` as "ball\_\_rotation".

Once the interval object has been declared as a new object (if it hasn't already been defined), then all of the arguments passed in are stored in that object to be used to calculate and perform the animation. These include the clip, start, and end values of the tween and the property to animate. In addition, we calculate the amount the property must change over the course of the animation and determine exactly how many intervals must be called in order to perform the animation over the course of the specified time, stored in the property `totalInts`.

How is this `totalInts` calculated exactly? First, you'll see that it uses a static property of Tweener, `intervalTime`. Let's go ahead and declare this, along with `__intervals`, at the top so we can see how the calculation works.

```

class com.oop.animation.Tweener extends Animator {

    private var __intervals:Object;
    static var intervalTime:Number = 40;

    function Tweener() {
        __intervals = {};
    }
}

```

`intervalTime` will dictate the frequency with which the IntervalManager calls the interval, in this case every 40 milliseconds. With that in mind, we can see that if a time of 2 seconds, or 2000 milliseconds, is passed to the `tween()` method, then the total number of intervals will be 50 (i.e., 2000/40). That means that the `runTween()` function should be called a total of 50 times in order to complete the animation. We'll keep track of the current interval through the interval object property count.

Finally, after storing a reference to the Tweener instance in the interval object, we call the IntervalManager's `IntervalManager()` method and pass in the interval object, instructing the IntervalManager to call `runTween()` on the Tweener instance (`this`) every 40 milliseconds (`Tweener.intervalTime`).

With all that work laid down, the `tweenTo()` method is considerably easier to pull off:

```
public function tweenTo(
    clip:Object,
    prop:String,
    destValue:Number,
    time:Number
):Void {
    tween(clip, prop, clip[prop], destValue, time);
}
```

`tweenTo()` will handle animating a clip from its current state to a new state. As such, a start value doesn't have to be passed in. However, after that point, the functionality behind the scenes is exactly that of `tween()`, so all we do is pass the values on to `tween()` with the clip's current property value being passed as the start value.

Before we get to actually performing the tween, let's add the ability to stop the tween, if needed. This is simply done by calling `clearInterval()` on the `IntervalManager`. Since the interval object being used is stored in our `Tweener` instance using the clip's name and property, that is all we need passed in to halt the interval.

```
public function haltTween(clip:Object, prop:String) {
    IntervalManager.clearInterval(
    ↪ __intervals[clip._name + "_" + prop], "interval");
}
```

All that's left now is to determine how the tween will be performed in `runTween()`. To start off easily, we'll perform a linear tween, meaning that the value of the clip's property will change an equal amount each interval. Let's add this to the `runTween()` method and see how it works.

```
private function runTween(intObj:Object):Void {
    var startProp:Number = intObj.startProp;
    var changeProp:Number = intObj.changeProp;
    var count:Number = intObj.count;
    var total:Number = intObj.totalInts;
    intObj.clip[intObj.prop] = startProp + ((changeProp/total)*count);
    var eventName:String = "tween";
    if (intObj.count++ >= intObj.totalInts) {
        intObj.clip[intObj.prop] = intObj.endProp;
        IntervalManager.clearInterval(intObj, "interval");
        eventName = "endTween";
    }
    dispatchEvent(eventName, {clip:intObj.clip, prop:intObj.prop});
}
```

For better readability, the first four lines simply store the properties of the interval object used to calculate the new property value. This calculation is performed on the fifth line of the function where the clip's property (accessed using bracket notation) is given a new value determined by the current count. This formula works by taking the amount the

## OBJECT-ORIENTED ACTIONSCRIPT FOR FLASH 8

property needs to change each interval call (changeProp/total) and multiplying it by the current count, adding this to the initial value of the property. It's perhaps a bit easier to see how this formula works by plugging in numbers. For instance, if we were tweening a clip's `_x` position from 50 to 450 over the course of 2 seconds, then the changeProp value would be 400 and the totalInts would be 50 (2000 milliseconds / Tweener.intervalTime). This means after 1 second has passed, the count would be 25 (half of the total 2 seconds worth of 50 intervals). The formula would then become, after substitution

```
clip._x = 50 + ((400/50)*25);
```

So the clip would be placed at 250, halfway between 50 and 450.

The conditional that follows just checks to see whether the current interval, represented by count, equals or exceeds the allotted number of intervals for the animation. If so, the clip's property is given the destination value and the interval is cleared. The last line of code dispatches an event informing listeners either that a tween or an endTween has occurred.

That's the bare bones of our Tweener class, so let's try it out to see how it performs. Create a new Flash document and save it as `tweenerTest.fla`. Rename the default layer `clip`. Draw a circle on the stage, converting it to a movie clip symbol. Name the instance `circle_mc`, as shown in Figure 17-2.

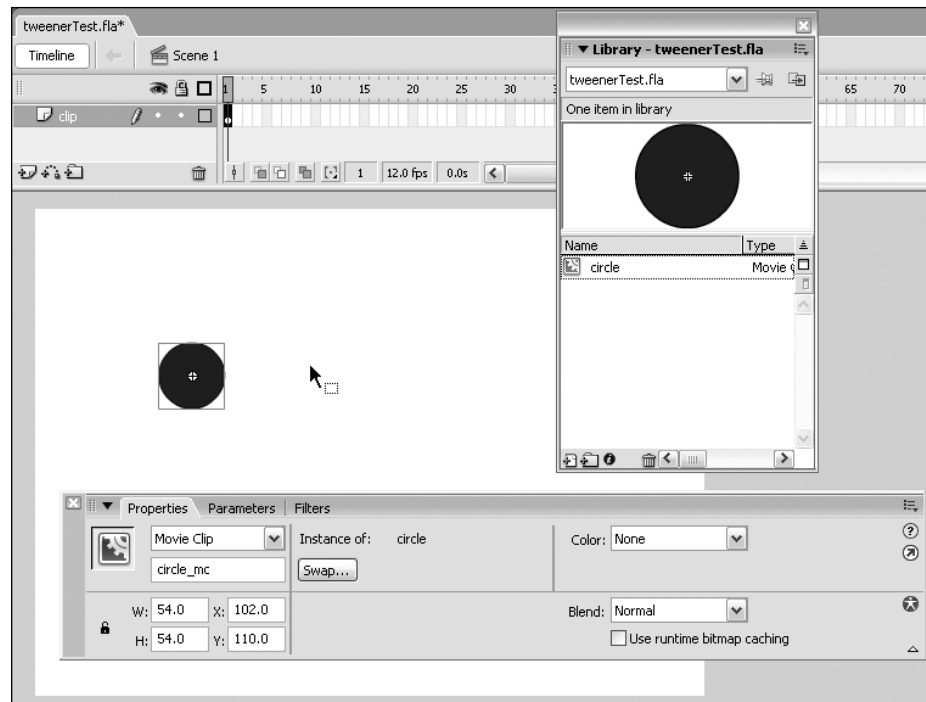


Figure 17-2. The stage set for testing Tweener

Create a new layer named code, and for its first (and only) frame, enter the following code into the ActionScript panel:

```
import com.oop.animation.Tweener;

init();
function init():Void {
    tweener = new Tweener();
    onMouseDown = tweenCircle;
}
function tweenCircle():Void {
    tweener.tweenTo(circle_mc, "_x", _xmouse, 1000);
    tweener.tweenTo(circle_mc, "_y", _ymouse, 1000);
}
```

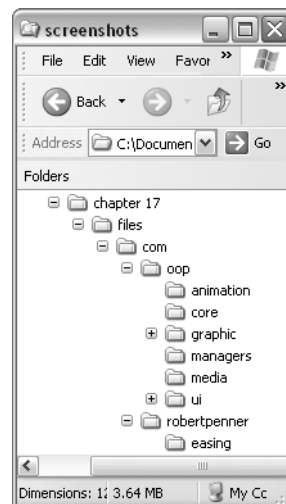
If you test the movie now, you'll see that the circle performs a linear tween to wherever the mouse is clicked. We use the `tweenTo()` method to animate the clip from its current `_x` and `_y` position to the mouse position over the course of 1 second. Because the Tweener class utilizes the IntervalManager, we don't have to worry about conflicting tweens or interval calls causing any animation irregularities.

What is missing, however, is any type of organic feel to the movement. We can add that in by using a bit of easing, or acceleration/deceleration, to the animated movement.

## Easer

It just so happens that someone has already spent the time to write the ActionScript equations that can be used to create easing movement, similar to what we did earlier with the linear animation equation. Robert Penner ([www.robertpenner.com](http://www.robertpenner.com)) has provided open source easing equations for several versions of Flash now, equations you can download from [www.robertpenner.com/easing/](http://www.robertpenner.com/easing/) or from this chapter's download files. We'll leverage this useful and excellent code to provide more interesting animation possibilities without a large amount of excess work.

The first thing to do is to install his classes into the same directory as your current class files. The package for the easing equations is `com.robertpenner.easing`, so you can copy the `robertpenner` directory right into the `com` directory of your own classes, as you see in Figure 17-3.



**Figure 17-3.** The path to the new `robertpenner.easing` package

## OBJECT-ORIENTED ACTIONSCRIPT FOR FLASH 8

Let's take a look at one of his easing classes, Cubic, reproduced here with kind permission from Mr. Penner.

```
class com.robertpenner.easing.Cubic {
    static function easeIn (t:Number, b:Number,
➤ c:Number, d:Number):Number {
        return c*(t/=d)*t*t + b;
    }
    static function easeOut (t:Number, b:Number,
➤ c:Number, d:Number):Number {
        return c*((t=t/d-1)*t*t + 1) + b;
    }
    static function easeInOut (t:Number, b:Number,
➤ c:Number, d:Number):Number {
        if ((t/=d/2) < 1) return c/2*t*t*t + b;
        return c/2*((t-=2)*t*t + 2) + b;
    }
}
```

Here you can see that a single easing equation type (Cubic) has three different static methods, `easeIn()`, `easeOut()`, and `easeInOut()`. Let's make some modification to our Tweener class so that we may use these equations in our animations.

Go back to the `Tweener.as` file and add the following bold lines:

```
public function tween(
    clip:Object,
    prop:String,
    startValue:Number,
    destValue:Number,
    time:Number,
easeFunction:Function
):Void {
    clip[prop] = startValue;
    var intName:String = clip._name + "_" + prop;
    if (__intervals[intName] == undefined) __intervals[intName] = {};
    var intObj:Object = __intervals[intName];
    intObj.count = 1;
    intObj.totalInts = Math.floor(time/TweenerTest.intervalTime);
    intObj.startProp = startValue;
    intObj.endProp = destValue;
    intObj.changeProp = intObj.endProp - intObj.startProp;
    intObj.clip = clip;
    intObj.prop = prop;
intObj.easeFunction = (easeFunction == undefined) ?
➤ com.robertpenner.easing.Linear.easeNone : easeFunction;
    intObj.tweener = this;
    IntervalManager.setInterval(intObj, "interval",
➤ this, "runTween", TweenerTest.intervalTime, intObj);
    runTween(intObj);
}
```

```

public function tweenTo(
    clip:Object,
    prop:String,
    destValue:Number,
    time:Number,
    easeFunction:Function
):Void {
    tween(clip, prop, clip[prop], destValue, time, easeFunction);
}

```

Here you can see that we now accept another parameter into both the `tween()` and `tweenTo()` methods, a function for the ease. We might wish in the future to modify the easing equations to inherit from a single `Ease` class so that we could type this argument accordingly, but for now we'll type it as `Function`. In the `tween()` method, we check to see whether a function has been passed in and, if not, we use the `Linear.easeNone` method, which you'll notice if you look at that particular method is the linear equation we originally coded ourselves.

Finally, we need to change the way we calculate the clip's property each interval in the `runTween()` method. Instead of using our single equation, it should use the function passed in.

```

private function runTween(intObj:Object):Void {
    var startProp:Number = intObj.startProp;
    var changeProp:Number = intObj.changeProp;
    var count:Number = intObj.count;
    var total:Number = intObj.totalInts;
    intObj.clip[intObj.prop] =
    ➔ intObj.easeFunction(count, startProp, changeProp, total);
    if (intObj.count++ >= intObj.totalInts) {
        intObj.clip[intObj.prop] = intObj.endProp;
        IntervalManager.clearInterval(intObj, "interval");
        dispatchEvent("endTween", {clip:intObj.clip, prop:intObj.prop});
    }
}

```

Go back to your `tweenerTest.fla` file and pass in an easing function to see how it works. This particular combination produces a nice effect.

```

import com.oop.animation.Tweener;
import com.robertpenner.easing.*;

init();
function init():Void {
    tweener = new Tweener();
    onMouseDown = tweenCircle;
}
function tweenCircle():Void {
    tweener.tweenTo(circle_mc, "_x", _xmouse, 1000, Cubic.easeOut);
    tweener.tweenTo(circle_mc, "_y", _ymouse, 1000, Cubic.easeIn);
}

```

## Testing the Tweener

Let's set up a little bit more of a structured environment through which to test. This will prove useful as we create more animation classes. Create a new ActionScript file and save it into the same directory as `tweenerTest.fla`. This class won't be within our class structure, as we'll simply be using it to test some of our code. Add the following lines, which is the entirety of the class:

```
import com.oop.ui.UIObject;
import com.oop.ui.Block;

class AnimationTest extends UIObject {

    private var __blockHolder:UIObject;

    function AnimationTest() {}

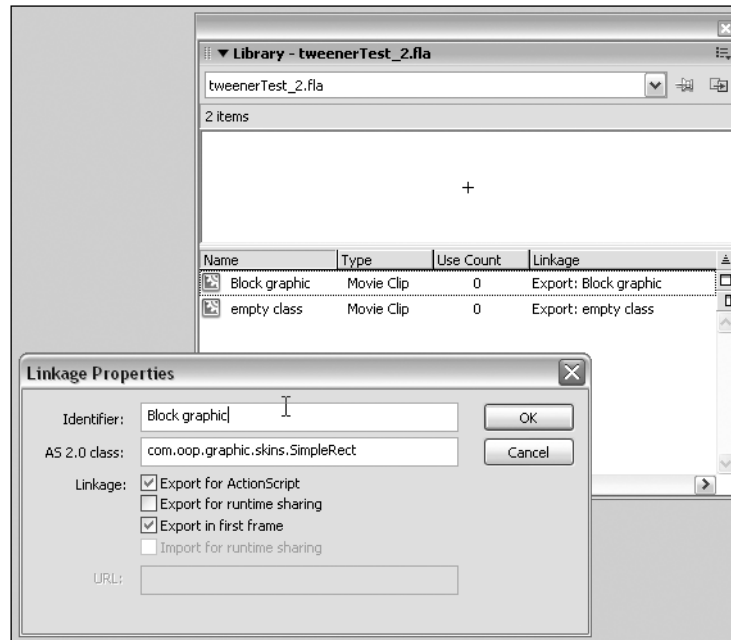
    private function init():Void {
        super.init();
        attachGraphics();
    }

    private function attachGraphics():Void {
        __blockHolder = createEmptyObject("__block", 0);
        __blockHolder.position =
    ➤ new flash.geom.Point(Stage.width/2, Stage.height/2);
        __blockHolder.createClassObject(Block, "__block", 0,
        {
            initialWidth:50,
            initialHeight:50,
            _x:-25,
            _y:-25
        }
        );
    }
}
```

Here we have a class that will draw an initial graphic, a `Block` instance, onto the stage so that we may animate it. In the `attachGraphics()` method, we create an empty `UIObject` named `__blockHolder`, and within this we create a `Block` instance named `__block`. The reason this is done is so that we may center the `Block` instance within `__blockHolder` (its default registration point is at the top left). This will be important if we were to test scaling and rotation animation, since the graphic would then transform from its center as opposed to the top-left corner.

Now create a new Flash document and save it into the same directory as `tweenerTest_2.fla`. In this file, create an empty movie clip symbol named `empty class` and export it as `empty`

class as well, and another movie clip symbol named and exported as Block graphic with its AS 2.0 class set as `com.oop.graphic.skins.SimpleRect`, as shown in Figure 17-4.



17

**Figure 17-4.** The two symbols needed for the animation test

In the only frame on the main timeline, add the following code:

```
Object.registerClass("empty class", AnimationTest);
attachMovie("empty class", "animationTest", 0);
```

If you test your movie now, you'll see a 50×50 gray box drawn onto the middle of the stage. Now let's try animating it!

Create a new ActionScript file and save it as `TweenerTest.as` into the same directory as `AnimationTest.as`. Add the following lines of code, which show the class in its entirety:

```
import com.oop.animation.Tweener;
import com.robertpenner.easing.*;

class TweenerTest extends AnimationTest {

    private var __tweener:Tweener;
    private var __positions:Array;
    private var __tweenCount:Number;
```

```

function TweenerTest() {}

private function init():Void {
    super.init();
    __tweener = new Tweener();
    __tweener.addEventListener("endTween", this, "startNextTween");
    __positions =
    [
        {x:100, y:100},
        {x:Stage.width-100, y:100},
        {x:Stage.width-100, y:Stage.height-100},
        {x:100, y:Stage.height-100}
    ];
    __tweenCount = 0;
    startNextTween();
}

private function startNextTween():Void {
    var pos:Object = __positions[__tweenCount];
    __tweener.tween(__blockHolder, "_rotation",
    ➔ 0, 90, 500, Quad.easeInOut);
    __tweener.tweenTo(__blockHolder, "_x", pos.x, 500, Quad.easeIn);
    __tweener.tweenTo(__blockHolder, "_y", pos.y, 500, Quad.easeOut);
    if (++__tweenCount >= __positions.length) __tweenCount = 0;
}
}

```

We are extending `AnimationTest` here so that we start off with the `Block` on the stage. In this class's `init()` method, we create a new `Tweener` instance and add this `TweenerTest` instance as a listener to the `endTween` event, which will call `startNextTween()` whenever it's fired. We then set four positions in the `__positions` object that we'll use in our tweens. At the end of the `init()`, we set the current tween to 0 (`__tweenCount`) and call `startNextTween()`, which should start everything rolling.

In `startNextTween()`, we determine the next position to tween to and store that in the local variable `pos`. We then set up three tweens, one for rotation and two for position. The rotation tween will go from 0 degrees to 90 degrees each tween, which will look fine considering we are simply rotating a plain gray rectangle. For the two coordinate positions, we use the current values stored in `pos`. To each of these three tweens, we pass a different easing equation.

The last line of `startNextTween()` increments `__tweenCount`. Once the value exceeds the number of positions, the count is set back to 0. In this way, the `Block` instance will continuously loop through the four positions.

Return to `tweenerTest_2.fla` and change the lines of code to read as follows:

```

Object.registerClass("empty class", TweenerTest);
attachMovie("empty class", "tweenerTest", 0);

```

Test the movie once more to see the Block performing a number of complex tweens about the stage, all managed by our animation classes. We'll continue to use this setup in our later animation experiments.

## Enhancing Tweener

17

A common need for animation in Flash is tweening the position of clips on the stage, which obviously involves changing not one, but two variables over the course of the animation. To accomplish this, the Tweener class could be used (as we did in the previous exercise), but it would be helpful to have a class that handled specifically the tweening of position. We'll do this in a moment with a new Mover class.

In order to create the Mover class, we'll first add a little enhancement to the Tweener class in order to make it more powerful and useful under more circumstances. Right now, the Tweener is limited to altering one property per call, and that is all done within the scope of the Tweener instance itself. When more complex properties need to be changed, like a Color object or even a multidimensional structure, the Tweener would fail. What we need then is a way to have the Tweener handle multiple and diverse values over the course of an animation. We'll accomplish this with a few new methods whose collective purpose is merely to calculate easing values for any type of object passed in, and then pass these values over the course of an animation to a user-defined method outside of the scope of the Tweener instance. In this way, we'll open up endless possible uses for the Tweener class, from tweening a movie clip's position to altering advanced Color transforms to even performing 3D transformations.

Open up the Tweener class, and add a new `callTween()` method to the public functions. This will be the method we would call to initiate this new type of interval call.

```
public function callTween(
    clip:Object,
    method:String,
    startParams:Object,
    destParams:Object,
    time:Number,
    easeFunction:Function
):Void {
    var intName:String = clip._name + "_" + prop;
    if (__intervals[intName] == undefined) __intervals[intName] = {};
    var intObj:Object = __intervals[intName];    intObj.count = 0;
    intObj.totalInts = Math.floor(time/Tweener.intervalTime);
    intObj.startParams = startParams;
    intObj.endParams = destParams;
    intObj.changeParams = {};
    for (var i:String in startParams) {
        intObj.changeParams[i] = destParams[i] - startParams[i];
    }
    intObj.easeFunction = (easeFunction == undefined) ?
    ➔ com.robertpenner.easing.Linear.easeNone : easeFunction;
    intObj.clip = clip;
```

```

        intObj.method = method;
        IntervalManager.setInterval(intObj, "interval",
    ➤ this, "runTweenCall", Tweener.intervalTime, intObj);
        dispatchEvent("startTween", {clip:clip, method:method});
        runTweenCall(intObj);
    }

```

The arguments passed to this function will be very similar to those passed to `tween()`. The changes to note are that instead of a property passed in to alter, a method is passed in that the Tweener instance will call every interval. Also, `startParams` and `destParams` are objects as opposed to scalar values.

The body of the function is also very similar to that of `tween()`. A new interval object is created and given all the arguments to store. The `changeParams` are calculated by running through them all in a loop and finding the differences. You may note that in this case the count is initialized at 0 instead of 1. We do this because in this method we have no way of knowing what to do with the start values of the tween, as they could apply to object properties or something more abstract. However, the Tweener instance doesn't need to know this since the external method that will be called should be set to handle all the numbers and know what to do with them. Instead, the Tweener instance simply sets the count to 0 and immediately calls the `runTweenCall()` method (the last line of the function) so that the external method might be invoked to handle the animation at the 0 count, which would be the tween's starting values.

Since this function and the `tween()` function both create the interval object in the exact same way, let's move that functionality into a separate method to clean things up. Add the following to the private methods:

```

private function getInterval(clip:Object, prop:String):Object {
    var intName:String = clip._name + "_" + prop;
    if (__intervals[intName] == undefined) __intervals[intName] = {};
    return __intervals[intName];
}

```

With this functionality now removed from the `tween()` and `callTween()` methods, we can replace the lines in those two places with a call to this method.

```

public function tween(
    clip:Object,
    prop:String,
    startValue:Number,
    destValue:Number,
    time:Number,
    easeFunction:Function
):Void {
    clip[prop] = startValue;
    var intObj:Object = getInterval(clip, prop);
    intObj.count = 1;
    intObj.totalInts = Math.floor(time/Tweener.intervalTime);
    ...
    etc.

```

```

public function callTween(
    clip:Object,
    method:String,
    startParams:Object,
    destParams:Object,
    time:Number,
    easeFunction:Function
):Void {
    var intObj:Object = getInterval(clip, method);
    intObj.count = 0;
    intObj.totalInts = Math.floor(time/Tweener.intervalTime);
    ...
    etc.

```

Now that the `getInterval()` is separate, its flaws may become apparent. For instance, the method assumes that the object passed in for the creation of the interval object will have a `_name` property. However, unless the Tweener always deals with movie clips, this won't be the case. A good example is the new Mover class we'll create, which, like the Tweener, will be an abstract object that doesn't exist on the stage and won't have a `_name` property like a movie clip. In such a circumstance, we need the Tweener class to be able to have a way to store intervals for these abstract objects. We'll do this by assigning a unique ID to any object that doesn't have a `_name` and using this value when creating an interval object.

The first thing we need is to create a way to assign these unique IDs for a Tweener instance. This will be accomplished with a new `__tweenID` property and a `getTweenID()` method.

```

class com.oop.animation.Tweener extends Animator {

    private var __intervals:Object;
    private var __tweenID:Number = 0
    static var intervalTime:Number = 40;

    function Tweener() {
        __intervals = {};
    }

    // PRIVATE
    // _____

    private function getTweenID():String {
        return String(++__tweenID);
    }

```

Each ID requested will increment the `__tweenID` count by one. We also return the ID as a string since the interval object names are stored as strings and not numbers.

The next step would be to alter the `getInterval()` method to handle objects with no `_name` and instead assign a `tweenID`.

```

private function getInterval(clip:Object, prop:String):Object {
    var clipName:String = clip._name || clip._$tweenID;
    if (clipName == undefined) {
        clipName = clip._$tweenID = getTweenID();
    }
    var intName:String = clipName + "_" + prop;
    if (__intervals[intName] == undefined) __intervals[intName] = {};
    return __intervals[intName];
}

```

So now instead of immediately assuming a `_name` exists, the first line assigns the `clip._name` to a local variable `clipName` or a special property named `_$tweenID` if the `clip._name` doesn't exist. On the second line, if `clipName` is undefined, we know that the clip doesn't have a `_name` property and hasn't yet been assigned a special `_$tweenID`. In this case, we call `getTweenID()` and assign its return value to both the `clipName` variable and the `clip._$tweenID` property, which can then be accessed the next time this object creates a tween. In the third line, the `clipName` variable is then used to create the name of the interval object.

Of course, with this change in how the interval is stored, we need to alter the `haltTween()` call as well. Make the following bold changes:

```

public function haltTween(clip:Object, prop:String) {
    IntervalManager.clearInterval(getInterval(clip, prop), "interval");
}

```

The final step to add our more robust functionality to Tweeners is to write the `runTweenCall()` method present in the `callTween()` code. This is the method that will be called every interval when the `callTween()` method begins a tween. In this method, the multiple values passed in to `callTween()` will have their new values calculated, and the external method will be called with these new values passed.

```

private function runTweenCall(intObj:Object):Void {
    var startParams:Number = intObj.startParams;
    var changeParams:Number = intObj.changeParams;
    var count:Number = intObj.count;
    var total:Number = intObj.totalInts;
    var p:Object = {};
    for (var i:String in changeParams) {
        p[i] = intObj.easeFunction(count,
➤ startParams[i], changeParams[i], total);
    }
    if (intObj.count++ > intObj.totalInts) {
        intObj.clip[intObj.method](intObj.endParams);
        IntervalManager.clearInterval(intObj, "interval");
        dispatchEvent("endTween",
➤ {clip:intObj.clip, method:intObj.method});
    } else {

```

```

        intObj.clip[intObj.method](p);
        dispatchEvent("tween", {clip:intObj.clip, method:intObj.method});
    }
}

```

Much like `callTween()` was very similar to `tween()`, `runTweenCall()` has much in common with `runTween()`. Within the method, the values within `changeParams` are run through, and the easing equation is used to calculate their new values. Once that is complete, we determine whether the number of intervals allotted for the tween is up, and if so, the `endParams` are passed to the external method. If not, then the changed params for this time in the interval are passed to the external method.

17

With Tweeners as complete as we need it for these exercises (you could certainly add more functionality such as the ability to loop or create random fluctuations in motion), here is the entire listing to check your code against:

```

import com.oop.animation.Animator;
import com.oop.managers.IntervalManager;

class com.oop.animation.Tweener extends Animator {

    private var __intervals:Object;
    private var __tweenID:Number = 0
    static var intervalTime:Number = 40;

    function Tweener() {
        __intervals = {};
    }

    // PRIVATE
    // _____

    private function getTweenID():String {
        return String(++__tweenID);
    }

    private function runTween(intObj:Object):Void {
        var startProp:Number = intObj.startProp;
        var changeProp:Number = intObj.changeProp;
        var count:Number = intObj.count;
        var total:Number = intObj.totalInts;
        intObj.clip[intObj.prop] =
        ➔ intObj.easeFunction(count, startProp, changeProp, total);
        var eventName:String = "tween";
        if (intObj.count++ >= intObj.totalInts) {
            intObj.clip[intObj.prop] = intObj.endProp;
            IntervalManager.clearInterval(intObj, "interval");
            eventName = "endTween";
        }
    }
}

```

```

    }
    dispatchEvent(eventName, {clip:intObj.clip, prop:intObj.prop});
}

private function runTweenCall(intObj:Object):Void {
    var startParams:Number = intObj.startParams;
    var changeParams:Number = intObj.changeParams;
    var count:Number = intObj.count;
    var total:Number = intObj.totalInts;
    var p:Object = {};
    for (var i:String in changeParams) {
        p[i] = intObj.easeFunction(count,
➤ startParams[i], changeParams[i], total);
    }
    if (intObj.count++ >= intObj.totalInts) {
        intObj.clip[intObj.method](intObj.endParams);
        IntervalManager.clearInterval(intObj, "interval");
        dispatchEvent("endTween",
➤ {clip:intObj.clip, method:intObj.method});
    } else {
        intObj.clip[intObj.method](p);
        dispatchEvent("tween", {clip:intObj.clip, method:intObj.method});
    }
}

private function getInterval(clip:Object, prop:String):Object {
    var clipName:String = clip._name || clip._$tweenID;
    if (clipName == undefined) {
        clipName = clip._$tweenID = getTweenID();
    }
    var intName:String = clipName + "_" + prop;
    if (__intervals[intName] == undefined) __intervals[intName] = {};
    return __intervals[intName];
}

// PUBLIC
// _____

public function haltTween(clip:Object, prop:String) {
    IntervalManager.clearInterval(getInterval(clip, prop), "interval");
}

public function tween(
    clip:Object,
    prop:String,
    startValue:Number,
    destValue:Number,
    time:Number,
    easeFunction:Function

```

```

):Void {
    clip[prop] = startValue;
    var intObj:Object = getInterval(clip, prop);
    intObj.count = 1;
    intObj.totalInts = Math.floor(time/Tweener.intervalTime);
    intObj.startProp = startValue;
    intObj.endProp = destValue;
    intObj.changeProp = intObj.endProp - intObj.startProp;
    intObj.clip = clip;
    intObj.prop = prop;
    intObj.easeFunction = (easeFunction == undefined) ?
    ➔ com.robertpenner.easing.Linear.easeNone : easeFunction;
    IntervalManager.setInterval(intObj, "interval",
    ➔ this, "runTween", Tweener.intervalTime, intObj);
    runTween(intObj);
    dispatchEvent("startTween", {clip:clip, prop:prop});
}

public function tweenTo(
    clip:Object,
    prop:String,
    destValue:Number,
    time:Number,
    easeFunction:Function
):Void {
    tween(clip, prop, clip[prop], destValue, time, easeFunction);
}

public function callTween(
    clip:Object,
    method:String,
    startParams:Object,
    destParams:Object,
    time:Number,
    easeFunction:Function
):Void {
    var intObj:Object = getInterval(clip, method);
    intObj.count = 0;
    intObj.totalInts = Math.floor(time/Tweener.intervalTime);
    intObj.startParams = startParams;
    intObj.endParams = destParams;
    intObj.changeParams = {};
    for (var i:String in startParams) {
        intObj.changeParams[i] = destParams[i] - startParams[i];
    }
    intObj.easeFunction = (easeFunction == undefined) ?
    ➔ com.robertpenner.easing.Linear.easeNone : easeFunction;
    intObj.clip = clip;
    intObj.method = method;

```

```

        IntervalManager.setInterval(intObj, "interval",
➤ this, "runTweenCall", Tweener.intervalTime, intObj);
        dispatchEvent("startTween", {clip:clip, method:method});
        runTweenCall(intObj);
    }
}

```

## Mover

Now that we have added the extra functionality to Tweener, creating unique animators that can handle complex objects is pretty simple. Take for instance the Mover class discussed at the beginning of the last section. Let's look at the code necessary to create such a class using our Tweener.

Create a new ActionScript file and save it as `Mover.as` into the `com/oop/animation` directory where Tweener resides. Add the following lines of code, which is the framework of a simple Mover class:

```

import com.oop.animation.Animator;
import com.oop.animation.Tweener;

class com.oop.animation.Mover extends Animator {

    private var __tweener:Tweener;
    private var __clip:MovieClip;

    function Mover() {
        __tweener = new Tweener();
    }

    // PRIVATE
    // _____

    private function runMoveTween(intObj:Object):Void {
    }

    // PUBLIC
    // _____

    public function haltTween(clip:Object) {
    }

    public function tween(
        clip:MovieClip,
        startParams:Object,
        endParams:Object,
        time:Number,
        easeFunction:Function

```

```

):Void {
}

public function tweenTo(
    clip:MovieClip,
    endParams:Object,
    time:Number,
    easeFunction:Function
):Void {
}
}

```

As you can see, Mover isn't a Tweener itself, but instead has a Tweener instance that it will use to calculate all of the changed values. `tween()` and `tweenTo()` will initiate a tween, `runMoveTween()` will be the external method called by the Tweener instance, and `haltTween()` will end the tween when requested. Of course, the methods won't yet do anything, so let's fill them in.

```

public function tween(
    clip:MovieClip,
    startParams:Object,
    endParams:Object,
    time:Number,
    easeFunction:Function
):Void {
    __tweener.addEventListener("endTween", this);
    __clip = clip;
    dispatchEvent("startTween", {clip:clip});
    __tweener.callTween(this, "runMoveTween",
➔ startParams, endParams, time, easeFunction);
}

public function tweenTo(
    clip:MovieClip,
    endParams:Object,
    time:Number,
    easeFunction:Function
):Void {
    tween(clip, {x:clip._x, y:clip._y}, endParams, time, easeFunction);
}

```

With the Tweener instance providing most of the functionality, setting up a tween of multiple properties is relatively easy. Note that in the `tween()` method, we add the Mover instance as a listener to the Tweener instance. This means that we'll need to define an `endTween()` method that will be called when the `endTween` event is dispatched. The remaining lines in the method save a reference to the clip being tweened, dispatch the `startTween` event, and set up `__tweener` to call `runMoveTween()` at every tween interval. The Tweener instance will handle calculating what values should be sent to this method by using the `startParams`, the `endParams`, and the easing function passed in.

## OBJECT-ORIENTED ACTIONSCRIPT FOR FLASH 8

As before, `tweenTo()` merely needs to call `tween()` with the `startParams` passed in.

Next, let's flesh out the final two methods we defined earlier while adding the new `endTween()` method that will be called when `__tweener` completes a tween.

```
// PRIVATE
// _____

private function endTween(infoObj:Object):Void {
    __tweener.removeEventListener("endTween", this);
    dispatchEvent("endTween", {clip:__clip});
}

private function runMoveTween(intObj:Object):Void {
    __clip._x = intObj.x;
    __clip._y = intObj.y;
    dispatchEvent("tween");
}

// PUBLIC
// _____

public function haltTween(clip:Object) {
    __tweener.haltTween(this, "runMoveTween");
}
```

`endTween()` merely needs to remove the `Mover` instance as a listener for the `endTween` event from `__tweener`. `runMoveTween()`, which receives values calculated by `__tweener`, sets the clip's `_x` and `_y` properties to the current tween values. Finally, `haltTween()` calls the `haltTween()` method on `__tweener`.

The simple `Mover` class is complete, so let's try it out. Create a new ActionScript file and save it into the same directory as `AnimationTest` and `TweenerTest` from the first section's exercise. Add the following code:

```
import com.oop.animation.Mover;
import com.robertpenner.easing.*;

class MoverTest extends AnimationTest {

    private var __mover:Mover;

    function MoverTest() {}

    private function init():Void {
        super.init();
        __mover = new Mover();
        onMouseDown = moveToMouse;
    }
}
```

```

        private function moveToMouse():Void {
            __mover.tweenTo(__blockHolder,
                ➔ {x:_xmouse, y:_ymouse}, 600, Circ.easeInOut);
        }
    }
}

```

Since `AnimationTest`, if you recall, draws a `Block` instance on the stage, `MoverTest` simply has to decide how to move it about. In the `init()` method, a new `Mover` instance is created and `moveToMouse()` is assigned to the `onMouseDown` handler. `moveToMouse()` calls a `tweenTo()` on the `Mover` instance, animating `__blockHolder` to the current mouse position using the `Circ.easeInOut` easing equation.

To test this out, duplicate the `tweenerTest.fla` file from the first section's exercise and save it as `moverTest.fla`. This file has an empty clip in the library and the `Block` graphic skin. Change the Frame 1 code on the main timeline to the following:

```

Object.registerClass("empty class", MoverTest);
attachMovie("empty class", "moverTest", 0);

```

Test the movie and click the stage to see the `Block` tween to the clicked position. Not bad! What would be great now is if we could add a little dynamic blur to the fast-moving object in order to create a little more realistic movement. Turns out that in Flash 8, we can!

## Motion blur

With the new `BlurFilter` available in Flash 8, we can add a bit of motion blur to our objects as they move across the stage, a cool little effect with minimal additional work. This won't be a true directional motion blur, but a close approximation. When things are moving fast across the screen in a blur, it will be hard to tell the difference anyway!

What we'll do is calculate the distance an object has traveled since the last interval call and use the distance on each axis to set the blur on each axis, properties of the `BlurFilter`. Therefore, if an object moves more vertically than horizontally, the blur on the y axis will be greater than the blur on the x axis.

The first thing we'll add to our `Mover.as` file is the code to import the `BlurFilter` class and set its default values for `Mover`.

```

import com.oop.animation.Animator;
import com.oop.animation.Tweener;
import flash.filters.BlurFilter;

class com.oop.animation.Mover extends Animator {

    private var __tweener:Tweener;
    private var __clip:MovieClip;
    public var blur:Boolean = false;
    public var blurFactor:Number = 1;
    public var blurQuality:Number = 1;
}

```

```
function Mover() {
    __tweener = new Tweener();
}
```

`blur` will determine whether the Mover uses the `BlurFilter` on its tweened clip, turned off by default. `blurFactor` will be used to help determine the `blurX` and `blurY` values for the `BlurFilter` (higher values will produce more blur), and `blurQuality` will be passed to the `BlurFilter` as its quality property, controlling how many blur calculations are performed on the object. Higher quality obviously produces better results, but at the cost of more processing cycles.

I have decided to leave these properties public for ease of demonstration here. Feel free to create getter/setter methods for private variables if you so wish. Also, there is the possibility of adding blur values for each individual tween as opposed to for every tween performed by a single Mover instance. That, again, is up to you.

The next step is to add a `BlurFilter` to the tweened clip when a tween starts (and when `blur` is set to true). Add these lines to the `tween()` method:

```
public function tween(
    clip:MovieClip,
    startParams:Object,
    endParams:Object,
    time:Number,
    easeFunction:Function
):Void {
    __tweener.addEventListener("endTween", this);
    __clip = clip;
    if (blur) {
        var f:Array = __clip.filters || [];
        if (f[f.length-1] instanceof BlurFilter) f = f.slice(0, -1);
        f.push(new BlurFilter(0, 0, blurQuality))
        __clip.filters = f;
    }
    dispatchEvent("startTween", {clip:clip});
    __tweener.callTween(this, "runMoveTween",
    ➤ startParams, endParams, time, easeFunction);
}
```

If `blur` is set to true, we grab either the `filters` array currently assigned to the object or, if it doesn't yet exist, create a new array. Then, if the last filter in this array is already a `BlurFilter` instance, we remove it. This assumes that clips passed to the Mover that should be blurred should not already have a `BlurFilter` instance applied. Finally, we push a new `BlurFilter` instance into the `filters` array with no blur yet applied (0 values for both the `blurX` and `blurY`) and reassign this modified array back as the clip's filters. We are now ready to tween!

Now that the `BlurFilter` is prepared on the tweened clip, we can alter its values each frame to create the motion blur. This will all occur in the `runMoveTween()` method.

```

private function runMoveTween(intObj:Object):Void {
    __clip._x = intObj.x;
    __clip._y = intObj.y;
    if (blur) {
        if (__clip._$moverLastX != undefined) {
            var f:Array = __clip.filters;
            var bf:BlurFilter = f[f.length-1];
            var factor:Number = blurFactor/10;
            bf.blurX = Math.abs((intObj.x - __clip._$moverLastX)*factor);
            bf.blurY = Math.abs((intObj.y - __clip._$moverLastY)*factor);
            __clip.filters = f;
        }
        __clip._$moverLastX = intObj.x;
        __clip._$moverLastY = intObj.y;
    }
    dispatchEvent("tween");
}

```

Here is the logic applied to create our blur. Each frame, the Mover modifies two special values on the tweened clip, `_$moverLastX` and `_$moverLastY`, so named to hopefully prevent any conflict with values already on the clip. These variables will hold the last screen position of the clip on each axis. The first time this method is called, these values don't exist, but every subsequent time we can check the difference of these values and the current position of the clip and use this to calculate the blur. How exactly is that calculated? A factor variable is determined based on the `blurFactor` property of Mover. The movement on each axis is multiplied by this factor to determine the blur amount, so the larger the factor and the greater the movement, the larger the blur. By default, `blurFactor` is set to 1, which makes factor resolve to 0.1. Therefore, with these settings, a movement of 50 pixels on the x axis will set the `BlurFilter`'s `blurX` value to 5. This calculation occurs every time this method is called during a tween to create a dynamic blur.

The last step would be to remove the blur upon completion of the tween. This is taken care of in the `haltTween()` method.

```

public function haltTween(clip:Object) {
    if (blur) {
        delete __clip._$moverLastX;
        delete __clip._$moverLastY;
        __clip.filters = __clip.filters.slice(0, -1);
    }
    __tweener.haltTween(this, "runMoveTween");
}

```

Once the tween is complete, the special variables used for the tween are deleted, and the `BlurFilter`, last in the list of filters for the clip, is removed.

To test this in action, return to the `MoverTest.as` file and set the Mover instance's `blur` property to true.

```
private function init():Void {
    super.init();
    __mover = new Mover();
    __mover.blur = true;
    onMouseDown = moveToMouse;
}
```

Test your movie, and you should see a subtle blur as you click the stage and tween the clip. To get a more distinct blur (less distinct?), try raising the `blur` factor.

```
private function init():Void {
    super.init();
    __mover = new Mover();
    __mover.blur = true;
    __mover.blurFactor = 3;
    onMouseDown = moveToMouse;
}
```

With just a little bit of code encapsulated in an animation class, we've now enabled the ability to tween the position of any movie clip object in our movies and have a motion blur applied!

## Transitioning views

This next group of classes, we'll explore the transitioning of visual objects from one state to another. By abstracting these transition classes out from the objects themselves, they may be reused over and over from one project to another with little to no additional work. For instance, if you were developing an image slideshow and you wanted a simple fade in/out of all the images, you might consider making a function that uses an `onEnterFrame` or an interval to change the alpha of one image till it disappeared, then tweens up the opacity of another image to bring it into view. This sort of transition is fairly easy to accomplish and might even be something that you have already copied and pasted from one project to another. How much easier would it be, even for this simple function, to create a transition instance, pass it the clip to be transitioned, and just tell it to start. Once completed, an event would be broadcast so you could have another transition tween in the new image. It becomes more obviously useful when dealing with more complex transitions, as we'll create through the rest of this chapter, using some of the bitmap manipulation available in Flash 8. By separating the transition code from the graphic objects themselves, these classes can be reused in any project with little fuss.

## Transition

All transitions will have similar needs, so it makes sense to have a class from which all transitions will inherit. This class will establish the common properties and the start and end events for transitions....

[Chapter continues. End of excerpt]