

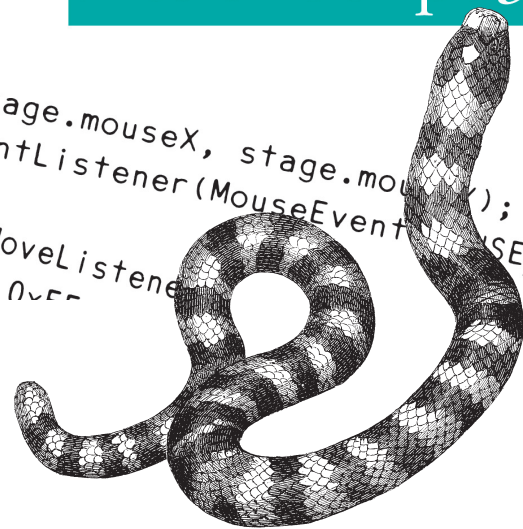
Adobe Presents

*Colin Moock's*

# ActionScript 3.0 From the Ground Up Tour



Essential  
ActionScript 3.0



O'REILLY



# Welcome

Welcome to the *ActionScript 3.0: From the Ground Up Tour!* In collaboration with Colin Moock, FITC Design and Technology Events, O'Reilly, and participating academic institutions around the world, Adobe is thrilled to bring you this world-class day of training. Following the tradition of Flex Camp (<http://flex.org/camp/>) and the onAIR bus tour (<http://onair.adobe.com/>), this lecture is an important part of Adobe's ongoing initiative to bring knowledge to the development community.

At Adobe, we understand that a tool is only useful when you know how to use it. And we're committed to helping you gain that knowledge. So sit back, get ready for a high-paced day of learning, and most of all have fun!

## Links and Resources

The entire set of notes for today's lecture are available at:

<http://moock.org/lectures/groundUpAS3>

The code for the virtual zoo application can be obtained at:

[http://moock.org/eas3/examples/moock\\_eas3\\_examples/virtualzoo\\_final](http://moock.org/eas3/examples/moock_eas3_examples/virtualzoo_final)

For a prose version of today's lecture in book form, see Colin Moock's *Essential ActionScript 3.0* (O'Reilly, 2007). Details about the book are included on the back cover of this program. For more books on Adobe technologies, see the Adobe Developer Library series, at <http://www.oreilly.com/store/series/adl.csp>.

## Agenda

9:00-10:30	First Steps: Programming tools, classes and objects, packages
10:30-10:45	Break
10:45-12:30	Variables, values, references, methods
12:30-1:30	Lunch Break
1:15-2:45	Conditionals, loops, Boolean logic, encapsulation, static members
2:45-3:00	In-Room Break
3:00-4:30	Functions, inheritance, compilation, type annotations, datatypes
4:30-4:45	Break
4:45-6:00	Events, display, image loading, text, interactivity

# Key Learning

The following table lists some of today's most important concepts.

Concept	Example
<i>Classes</i> are blueprints for objects.	<pre>class VirtualPet { }</pre>
<i>Objects</i> (or <i>instances</i> ) are the things in a program, such as a number, a car, a button, a point in time	<pre>new VirtualPet()</pre>
Some classes are built into ActionScript, others are custom-made.	<i>MovieClip, TextField, Sound, String</i>
A <i>package</i> contains a class so its name doesn't conflict with other names.	<pre>package zoo {     class VirtualPet {     } }</pre>
A <i>constructor method</i> initializes objects.	<pre>class VirtualPet {     public function VirtualPet () {         // Initialization code     } }</pre>
A <i>function</i> is a reusable set of instructions.	<pre>function doSomething () {     // Instructions go here }</pre>
A function can generate a result known as a <i>return value</i> .	<pre>function square (x) {     return x * x; }</pre>
A <i>variable</i> is an <i>identifier</i> (name) associated with a <i>value</i> (an object).	<pre>var pet = new VirtualPet();</pre>
Variables <i>do not store values</i> ; they merely refer to them.	<pre>var firstReferenceToPet = new VirtualPet(); var secondReferenceToPet = firstReferenceToPet;</pre>
Local variables are temporary, used to track data within a method or function only.	<pre>function square (x) {     var total = x * x;     return total; }</pre>



Concept	Example
<i>Instance variables</i> describe an object's characteristics	<pre>class VirtualPet {     private var petName;     private var currentCalories; }</pre>
<i>Instance methods</i> define an object's behaviors (things it can do)	<pre>class VirtualPet {     public function eat (numberOfCalories) {     } }</pre>
<i>Static variables</i> track information relating to entire class	<pre>class VirtualPet {     private static var maxNameLength = 20; }</pre>
<i>Static methods</i> define behaviors relating to entire class	Math.min(4, 5)
An <i>argument</i> is a value passed to a method or function	eat( <b>100</b> );
A <i>parameter</i> is a local variable defined in a method or function header, whose value is set via an argument	<pre>public function eat (<b>numberOfCalories</b>) { }</pre>
The <i>private</i> modifier means access within this class only; <i>protected</i> means this class or subclasses; <i>internal</i> means this package; <i>public</i> means anywhere	<pre>internal class VirtualPet {     private var petName;     public function eat (numberOfCalories) {     } }</pre>
A <i>conditional</i> is a statement that executes once when a condition is met	<pre>if (testExpression) {     codeBlock1 } else {     codeBlock2 }</pre>
A <i>loop</i> is a statement that executes for as long as a condition is met	<pre>while (testExpression) {     codeBlock }</pre>

Concept	Example
In an instance method, <i>this</i> means the object through which the method was invoked	<pre>public function eat (numberOfCalories) {     this.currentCalories += numberOfCalories; }</pre>
In an constructor method, <i>this</i> means the object being initialized	<pre>public function VirtualPet () {     this.petName = "Stan"; }</pre>
<i>Inheritance</i> is a relationship between two or more classes where one borrows (or <i>inherits</i> ) the variable and method definitions of another	<pre>public class Food { } public class Apple extends Food { }</pre>
A <i>datatype</i> is a set of values. Every class defines a datatype: the set of instances of that class, plus instances of descendant classes.	—
Type annotations tell the compiler the datatype of a variable, parameter, or function return value.	<pre>class VirtualPet {     private var petName:<b>String</b>; }</pre>
The compiler uses type annotations to detect <i>reference errors</i> at compile time.	1061: Call to a possibly undefined method eatt through a reference with static type zoo: VirtualPet.
A <i>cast operation</i> tells the compiler the datatype of a single value, and can result in a runtime type conversion.	Apple(foodItem).hasWorm()
Every <i>.swf</i> file must have a main class.	—
A <i>.swf</i> file's main class must inherit from either <i>Sprite</i> or <i>MovieClip</i> .	<pre>public class VirtualZoo <b>extends Sprite</b> { }</pre>
At runtime, Flash creates an instance of the <i>.swf</i> file main class automatically, then adds it to the screen (i.e., puts it on the <i>display list</i> ).	—
The <i>display list</i> is the hierarchy of all objects currently eligible for screen display in Flash Player.	—

Concept	Example
Key classes for displaying things include: <ul style="list-style-type: none"> <li>• <i>DisplayObject</i>: base display class</li> <li>• <i>InteractiveObject</i>: adds mouse/keyboard functionality</li> <li>• <i>DisplayObjectContainer</i>: adds containment for grouping</li> <li>• <i>Sprite</i>: adds dragability, button-style interaction features</li> <li>• <i>MovieClip</i>: adds timeline control</li> </ul>	—
To put an object on screen, use <i>addChild()</i> and <i>removeChild()</i> .	<code>graphicsContainer.addChild(hungryIcon);</code>
To load an external display asset, use the <i>Loader</i> class.	<pre>petAlive = new Loader(); petAlive.load(new URLRequest(     "pet-alive.gif"));</pre>
<i>Event dispatching</i> is a system for one object to tell other objects that “something happened” <ul style="list-style-type: none"> <li>• <i>event</i>: the thing that happened</li> <li>• <i>event target</i>: the object to which the “event” pertains</li> <li>• <i>event listeners</i>: methods that register to be notified of the “event”</li> </ul>	—
To register for an event, use <i>addEventListener()</i> .	<code>appleBtn.addEventListener(MouseEvent.CLICK, appleBtnClick);</code>

---

Remember, learning to program is a life-long process.

---

## ActionScript Overview

ActionScript 3.0 is an object-oriented language for creating applications and scripted multimedia content for playback in Flash client runtimes (such as Flash Player and Adobe AIR). With a syntax reminiscent of Java and C#, ActionScript’s core language should be familiar to experienced programmers. For example, the following code creates a variable named `width`, of type *int* (meaning integer), and assigns it the value 25:

```
var width:int = 25;
```

The following code creates a *for* loop that counts up to 10:

```
for (var i:int = 1; i <= 10; i++) {
    // Code here runs 10 times
}
```

And the following code creates a class named *Product*:

```
// The class definition
public class Product {
    // An instance variable of type Number
    var price:Number;

    // The Product class constructor method
    public function Product () {
        // Code here initializes Product instances
    }

    // An instance method
    public function doSomething ():void {
        // Code here executes when doSomething() is invoked
    }
}
```

## The Core Language

ActionScript 3.0’s core language is based on the ECMAScript 4th edition language specification, which is still under development as of October 2007.

---

The ECMAScript 4 specification can be viewed at <http://developer.mozilla.org/es4/spec/spec.html>. The ActionScript 3.0 specification can be viewed at <http://livedocs.macromedia.com/specs/actionscript/3>.

---

In the future, ActionScript is expected to be a fully conforming implementation of ECMAScript 4. Like ActionScript, the popular web browser language JavaScript is also based on ECMAScript. The future Firefox 3.0 web browser is expected to implement JavaScript 2.0 using the same code base as ActionScript, which was contributed to the Mozilla Foundation by Adobe in November, 2006 (for information, see <http://www.mozilla.org/projects/tamarin>).

ECMAScript 4 dictates ActionScript’s basic syntax and grammar—the code used to create things such as expressions, statements, variables, functions, classes, and objects. ECMAScript 4 also defines a small set of built-in datatypes for working with common values (such as *String*, *Number*, and *Boolean*).

Some of ActionScript 3.0’s key core-language features include:

- First-class support for common object-oriented constructs, such as classes, objects, and interfaces
- Single-threaded execution model
- Runtime type-checking
- Optional compile-time type-checking
- Dynamic features such as runtime creation of new constructor functions and variables
- Runtime exceptions
- Direct support for XML as a built-in datatype
- Packages for organizing code libraries
- Namespaces for qualifying identifiers
- Regular expressions

All Flash client runtimes that support ActionScript 3.0 share the features of the core language in common.

## Flash Runtime Clients

ActionScript programs can be executed in three different client runtime environments: Adobe AIR, Flash Player, and Flash Lite.

### Adobe AIR

Adobe AIR runs Flash-platform applications intended for desktop deployment. Adobe AIR supports SWF-format content, as well as content produced with HTML and JavaScript. Adobe AIR must be installed directly on the end-user’s computer at the operating-system level. For more information, see <http://www.adobe.com/go/air>.

### Flash Player

Flash Player runs Flash-platform content and applications intended for Web deployment. Flash Player is the runtime of choice for embedding SWF-format content on a web page. Flash Player is typically installed as a web browser add-on, but can also run in standalone mode.

### Flash Lite

Flash Lite runs Flash-platform content and applications intended for mobile-device deployment. Due to the performance limitations of mobile devices, Flash Lite typically lags behind Flash Player and Adobe AIR in both speed and feature set. As of October 2007, Flash Lite does not yet support ActionScript 3.0.

The preceding Flash client runtimes offer a common core set of functionality, plus a custom set of features that cater to the capabilities and security requirements of the runtime environment. For example, Adobe AIR, Flash Player, and Flash Lite all use the same syntax for creating a variable, but Adobe AIR includes window-management and filesystem APIs, Flash Lite can make a phone vibrate, and Flash Player imposes special web-centric security restrictions to protect the end-user’s privacy.

## Runtime APIs

Each Flash client runtime offers its own built-in set of functions, variables, classes, and objects—known as its *runtime API*. Each Flash client runtime’s API has its own name. For example, the Flash client runtime API defined by Flash Player is known as the *Flash Player API*.

All Flash client runtime APIs share a core set of functionality in common. For example, every Flash client runtime uses the same basic set of classes for displaying content on screen and for dispatching events.

Key features shared by all Flash client runtime APIs include:

- Graphics and video display
- A hierarchical event architecture
- Text display and input
- Mouse and keyboard control
- Network operations for loading external data and communicating with server-side applications
- Audio playback
- Printing
- Communicating with external local applications
- Programming utilities

## Components

In addition to the Flash client runtime APIs, Adobe also offers two different sets of *components* for accomplishing common programming tasks and building user interfaces. Flex Builder 2 and the free Flex 2 SDK include the *Flex framework*, which defines a complete set of user interface controls, such as *RadioButton*, *CheckBox*, and *List*. The Flash authoring tool provides a similar set of user interface components. The Flash authoring tool’s components combine code with manually created graphical assets that can be customized by Flash developers and designers.

Both the Flex framework and the Flash authoring tool’s component set are written entirely in ActionScript 3.0. The user interface components in the Flex framework generally have more features than those in the Flash authoring tool’s component set and, therefore, also have a larger file size.

---

User interface components from the Flex framework cannot be used in the Flash authoring tool, but user interface components from the Flash authoring tool *can* be used (both legally and technically) with Flex Builder 2 and mxmllc.

---

## The Flash File Format (SWF)

ActionScript code must be compiled into a *.swf* file for playback in one of Adobe's Flash client runtimes. A *.swf* file can include both ActionScript bytecode and embedded assets (graphics, sound, video, and fonts). Some *.swf* files contain assets only, and no code, while others contain code only, and no assets. A single ActionScript program might reside entirely within a single *.swf* file, or it might be broken into multiple *.swf* files. When a program is broken into multiple *.swf* files, one specific *.swf* file provides the program point of entry, and loads the other *.swf* files as required. Breaking a complex program into multiple *.swf* files makes it easier to maintain and, for Internet-delivered applications, can give the user faster access to different sections of the program.

## ActionScript Development Tools

Adobe offers the following tools for creating ActionScript code:

### Adobe Flash

<http://www.adobe.com/go/flash/>

A visual design and programming tool for creating multimedia content that integrates graphics, video, audio, animation, and interactivity. In Adobe Flash, developers create interactive content by combining ActionScript code with animation, manually created content, and embedded assets. Adobe Flash is also known as *the Flash authoring tool*. As of October 2007, the latest version of the Flash authoring tool is Flash CS3 (version 9 of the software).

### Adobe Flex Builder

<http://www.adobe.com/products/flex/productinfo/overview/>

A development tool for producing content using either pure ActionScript or *MXML*, an XML-based language for describing user interfaces. Flex Builder includes a development framework known as the Flex framework, which provides an extensive set of programming utilities and a library of skinnable, styleable user-interface controls. Based on Eclipse, the popular open-source programming tool, Flex Builder 2 can be used in either hand-coding mode or in a visual-development mode similar to Microsoft's Visual Basic.

### Adobe Flex 2 SDK

[http://www.adobe.com/go/flex2\\_sdk](http://www.adobe.com/go/flex2_sdk)

A free command-line toolkit for creating content using either pure ActionScript 3.0 or *MXML*. The Flex 2 SDK includes the Flex framework and a command-line compiler, *mxmcl* (both of which are also included with Adobe Flex Builder 2). Using the Flex 2 SDK, developers can create content for free in the programming editor of their choice. (For a wide variety of open-source tools and utilities for ActionScript development, see <http://osflash.org>.)

## The Virtual Zoo Program Code

Example A-1 shows the code for the *VirtualZoo* class, the virtual zoo program's main class.

```
Example A-1. The VirtualZoo class

package {
    import flash.display.Sprite;
    import zoo.*;
    import flash.events.*;

    public class VirtualZoo extends Sprite {
        private var pet:VirtualPet;
        private var petView:VirtualPetView;

        // Constructor
        public function VirtualZoo () {
            pet = new VirtualPet("Stan");
            petView = new VirtualPetView(pet);
            petView.addEventListener(Event.COMPLETE, petViewCompleteListener);
        }

        public function petViewCompleteListener (e:Event):void {
            addChild(petView);
            pet.start();
        }
    }
}
```

Example A-2 shows the code for the *VirtualPet* class, whose instances represent pets in the zoo.

```
Example A-2. The VirtualPet class

package zoo {
    import flash.utils.*;
    import flash.events.*;

    public class VirtualPet extends EventDispatcher {
        public static const NAME_CHANGE:String = "NAME_CHANGE";
        public static const STATE_CHANGE:String = "STATE_CHANGE";

        public static const PETSTATE_FULL:int     = 0;
        public static const PETSTATE_HUNGRY:int   = 1;
        public static const PETSTATE_STARVING:int = 2;
        public static const PETSTATE_DEAD:int     = 3;

        private static var maxNameLength:int = 20;
        private static var maxCalories:int = 2000;
        private static var caloriesPerSecond:int = 100;
        private static var defaultName:String = "Unnamed Pet";

        private var petName:String;
        private var currentCalories:int;
        private var petState:int;
        private var digestTimer:Timer;

        public function VirtualPet (name:String):void {
            setName(name);
            setCalories(VirtualPet.maxCalories/2);
        }

        public function start ():void {
```

```

digestTimer = new Timer(1000, 0);
digestTimer.addEventListener(TimerEvent.TIMER, digestTimerListener);
digestTimer.start();
}

public function stop ():void {
    if (digestTimer != null) {
        digestTimer.stop();
    }
}

public function setName (newName:String):void {
    if (newName.indexOf(" ") == 0) {
        throw new VirtualPetNameException();
    } else if (newName == "") {
        throw new VirtualPetInsufficientDataException();
    } else if (newName.length > VirtualPet.maxNameLength) {
        throw new VirtualPetExcessDataException();
    }

    petName = newName;
    dispatchEvent(new Event(VirtualPet.NAME_CHANGE));
}

public function getName ():String {
    if (petName == null) {
        return VirtualPet.defaultName;
    } else {
        return petName;
    }
}

public function eat (foodItem:Food):void {
    if (petState == VirtualPet.PETSTATE_DEAD) {
        trace(getName() + " is dead. You can't feed it.");
        return;
    }

    if (foodItem is Apple) {
        if (Apple(foodItem).hasWorm()) {
            trace("The " + foodItem.getName() + " had a worm. " + getName()
                + " didn't eat it.");
            return;
        }
    }

    trace(getName() + " ate the " + foodItem.getName()
        + " (" + foodItem.getCalories() + " calories).");
    setCalories(getCalories() + foodItem.getCalories());
}

private function setCalories (newCurrentCalories:int):void {
    if (newCurrentCalories > VirtualPet.maxCalories) {
        currentCalories = VirtualPet.maxCalories;
    } else if (newCurrentCalories < 0) {
        currentCalories = 0;
    } else {
        currentCalories = newCurrentCalories;
    }
}

```

```

var caloriePercentage:int = Math.floor(getHunger()*100);

trace(getName() + " has " + currentCalories + " calories"
    + " (" + caloriePercentage + "% of its food) remaining.");

if (caloriePercentage == 0) {
    if (getPetState() != VirtualPet.PETSTATE_DEAD) {
        die();
    }
} else if (caloriePercentage < 20) {
    if (getPetState() != VirtualPet.PETSTATE_STARVING) {
        setPetState(VirtualPet.PETSTATE_STARVING);
    }
} else if (caloriePercentage < 50) {
    if (getPetState() != VirtualPet.PETSTATE_HUNGRY) {
        setPetState(VirtualPet.PETSTATE_HUNGRY);
    }
} else {
    if (getPetState() != VirtualPet.PETSTATE_FULL) {
        setPetState(VirtualPet.PETSTATE_FULL);
    }
}

public function getCalories ():int {
    return currentCalories;
}

public function getHunger ():Number {
    return currentCalories / VirtualPet.maxCalories;
}

private function die ():void {
    stop();
    setPetState(VirtualPet.PETSTATE_DEAD);
    trace(getName() + " has died.");
}

private function digest ():void {
    trace(getName() + " is digesting...");
    setCalories(getCalories() - VirtualPet.caloriesPerSecond);
}

private function setPetState (newState:int):void {
    if (newState == petState) {
        return;
    }

    petState = newState;
    dispatchEvent(new Event(VirtualPet.STATE_CHANGE));
}

public function getPetState ():int {
    return petState;
}

private function digestTimerListener (e:TimerEvent):void {
    digest();
}
}
}

```

Example A-3 shows the code for the *Food* class, the superclass of the various types of food that pets eat.

*Example A-3. The Food class*

```
package zoo {
    public class Food {
        private var calories:int;
        private var name:String;

        public function Food (initialCalories:int) {
            setCalories(initialCalories);
        }

        public function getCalories ():int {
            return calories;
        }

        public function setCalories (newCalories:int):void {
            calories = newCalories;
        }

        public function getName ():String {
            return name;
        }

        public function setName (newName:String):void {
            name = newName;
        }
    }
}
```

Example A-4 shows the code for the *Apple* class, which represents a specific type of food that pets eat.

*Example A-4. The Apple class*

```
package zoo {
    public class Apple extends Food {
        private static var DEFAULT_CALORIES:int = 100;
        private var wormInApple:Boolean;

        public function Apple (initialCalories:int = 0) {
            if (initialCalories <= 0) {
                initialCalories = Apple.DEFAULT_CALORIES;
            }
            super(initialCalories);
            wormInApple = Math.random() >= .5;
            setName("Apple");
        }

        public function hasWorm ():Boolean {
            return wormInApple;
        }
    }
}
```

Finally, Example A-5 shows the code for the *Sushi* class, which represents a specific type of food that pets eat.

*Example A-5. The Sushi class*

```
package zoo {
    public class Sushi extends Food {
        private static var DEFAULT_CALORIES:int = 500;

        public function Sushi (initialCalories:int = 0) {
            if (initialCalories <= 0) {
                initialCalories = Sushi.DEFAULT_CALORIES;
            }
            super(initialCalories);
            setName("Sushi");
        }
    }
}
```

Example A-6 shows the code for the *VirtualPetNameException* class, which represents an exception thrown when an invalid pet name is specified.

*Example A-6. The VirtualPetNameException class*

```
package zoo {
    public class VirtualPetNameException extends Error {
        public function VirtualPetNameException (
            message:String = "Invalid pet name specified.") {
            super(message);
        }
    }
}
```

Example A-7 shows the code for the *VirtualPetExcessDataException* class, which represents an exception thrown when an excessively long pet name is specified for a pet.

*Example A-7. The VirtualPetExcessDataException class*

```
package zoo {
    public class VirtualPetExcessDataException
        extends VirtualPetNameException {
        public function VirtualPetExcessDataException () {
            super("Pet name too long.");
        }
    }
}
```

Example A-8 shows the code for the *VirtualPetInsufficientDataException* class, which represents an exception thrown when an excessively short pet name is specified for a pet.

*Example A-8. The VirtualPetInsufficientDataException class*

```
package zoo {
    public class VirtualPetInsufficientDataException
        extends VirtualPetNameException {
        public function VirtualPetInsufficientDataException () {
            super("Pet name too short.");
        }
    }
}
```

Example A-9 shows the code for the *VirtualPetView* class, which graphically displays a *VirtualPet* instance.

*Example A-9. The VirtualPetView class*

```
package zoo {
    import flash.display.*;
    import flash.events.*;
    import flash.net.*;
    import flash.text.*;

    public class VirtualPetView extends Sprite {
        private var pet:VirtualPet;
        private var graphicsContainer:Sprite;

        private var petAlive:Loader; // The pet in its alive state
        private var petDead:Loader; // The pet in its alive state
        private var foodHungry:Loader; // An icon for the hungry state
        private var foodStarving:Loader; // An icon for the starving state
        private var petName:TextField; // Displays the pet's name

        private var appleBtn:FoodButton; // Button for feeding the pet an apple
        private var sushiBtn:FoodButton; // Button for feeding the pet sushi

        // Load completion detection
        static private var numGraphicsToLoad:int = 4;
        private var numGraphicsLoaded:int = 0;

        public function VirtualPetView (pet:VirtualPet) {
            this.pet = pet;
            pet.addEventListener(VirtualPet.NAME_CHANGE,
                petNameChangeListener);
            pet.addEventListener(VirtualPet.STATE_CHANGE,
                petStateChangeListener);
            createGraphicsContainer();
            createNameTag();
            createUI();
            loadGraphics();
        }

        private function createGraphicsContainer ():void {
            graphicsContainer = new Sprite();
            addChild(graphicsContainer);
        }

        private function createNameTag ():void {
            petName = new TextField();
            petName.defaultTextFormat = new TextFormat("_sans",14,0x006666,true);
            petName.autoSize = TextFieldAutoSize.CENTER;
            petName.selectable = false;
            petName.x = 250;
            petName.y = 20;
            addChild(petName);
        }
    }
}
```

```
private function createUI ():void {
    appleBtn = new FoodButton("Feed Apple");
    appleBtn.y = 170;
    appleBtn.addEventListener(MouseEvent.CLICK, appleBtnClick);
    addChild(appleBtn);

    sushiBtn = new FoodButton("Feed Sushi");
    sushiBtn.y = 190;
    sushiBtn.addEventListener(MouseEvent.CLICK, sushiBtnClick);
    addChild(sushiBtn);
}

private function disableUI ():void {
    appleBtn.disable();
    sushiBtn.disable();
}

private function loadGraphics ():void {
    petAlive = new Loader();
    petAlive.load(new URLRequest("pet-alive.gif"));
    petAlive.contentLoaderInfo.addEventListener(Event.COMPLETE,
        completeListener);
    petAlive.contentLoaderInfo.addEventListener(IOErrorEvent.IO_ERROR,
        ioErrorListener);

    petDead = new Loader();
    petDead.load(new URLRequest("pet-dead.gif"));
    petDead.contentLoaderInfo.addEventListener(Event.COMPLETE,
        completeListener);
    petDead.contentLoaderInfo.addEventListener(IOErrorEvent.IO_ERROR,
        ioErrorListener);

    foodHungry = new Loader();
    foodHungry.load(new URLRequest("food-hungry.gif"));
    foodHungry.contentLoaderInfo.addEventListener(Event.COMPLETE,
        completeListener);
    foodHungry.contentLoaderInfo.addEventListener(IOErrorEvent.IO_ERROR,
        ioErrorListener);

    foodHungry.x = 15;
    foodHungry.y = 100;

    foodStarving = new Loader();
    foodStarving.load(new URLRequest("food-starving.gif"));
    foodStarving.contentLoaderInfo.addEventListener(Event.COMPLETE,
        completeListener);
    foodStarving.contentLoaderInfo.addEventListener(IOErrorEvent.IO_ERROR,
        ioErrorListener);

    foodStarving.x = 15;
    foodStarving.y = 100;
}

private function petStateChangeListener (e:Event):void {
    if (pet.getPetState() == VirtualPet.PETSTATE_DEAD) {
        disableUI();
    }
    renderCurrentPetState();
}
}
```

```

private function renderCurrentPetState ():void {
    for (var i:int = graphicsContainer.numChildren-1; i >= 0; i--) {
        graphicsContainer.removeChildAt(i);
    }
    var state:int = pet.getPetState();

    switch (state) {
        case VirtualPet.PETSTATE_FULL:
            graphicsContainer.addChild(petAlive);
            break;

        case VirtualPet.PETSTATE_HUNGRY:
            graphicsContainer.addChild(petAlive);
            graphicsContainer.addChild(foodHungry);
            break;

        case VirtualPet.PETSTATE_STARVING:
            graphicsContainer.addChild(petAlive);
            graphicsContainer.addChild(foodStarving);
            break;

        case VirtualPet.PETSTATE_DEAD:
            graphicsContainer.addChild(petDead);
            break;
    }
}

private function petNameChangeListener (e:Event):void {
    renderCurrentPetName();
}

private function renderCurrentPetName ():void {
    petName.text = pet.getName();
}

private function appleBtnClick (e:MouseEvent):void {
    pet.eat(new Apple());
}

private function sushiBtnClick (e:MouseEvent):void {
    pet.eat(new Sushi());
}

private function completeListener (e:Event):void {
    numGraphicsLoaded++;
    if (numGraphicsLoaded == numGraphicsToLoad) {
        renderCurrentPetState();
        renderCurrentPetName();
        dispatchEvent(new Event(Event.COMPLETE));
    }
}

private function ioErrorListener (e:IOErrorEvent):void {
    trace("Load error: " + e);
}
}
}

```

Example A-10 shows the code for the *FoodButton* class, which represents a simple clickable-text button.

*Example A-10. The FoodButton class*

```

package zoo {
    import flash.display.*
    import flash.events.*;
    import flash.text.*;

    public class FoodButton extends Sprite {
        private var text:TextField;
        private var upFormat:TextFormat;
        private var overFormat:TextFormat;

        public function FoodButton (label:String) {
            buttonMode = true;
            mouseChildren = false;
            upFormat = new TextFormat("_sans",12,0x006666,true);
            overFormat = new TextFormat("_sans",12,0x009999,true);

            text = new TextField();
            text.defaultTextFormat = upFormat;
            text.text = label;
            text.autoSize = TextFieldAutoSize.CENTER;
            text.selectable = false;
            addChild(text);

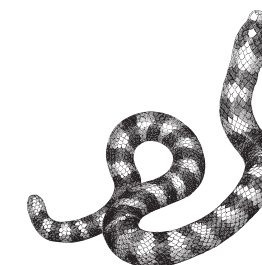
            addEventListener(MouseEvent.MOUSE_OVER, mouseOverListener);
            addEventListener(MouseEvent.MOUSE_OUT, mouseOutListener);
        }

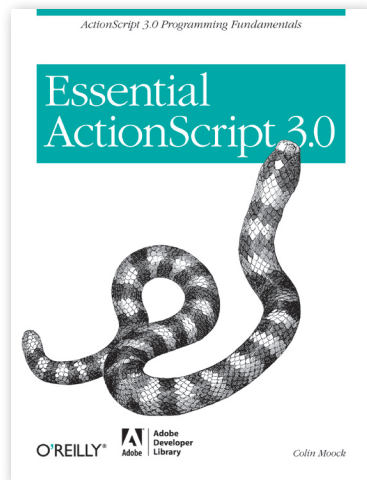
        public function disable ():void {
            mouseEnabled = false;
        }

        public function mouseOverListener (e:MouseEvent):void {
            text.setTextFormat(overFormat);
        }

        public function mouseOutListener (e:MouseEvent):void {
            text.setTextFormat(upFormat);
        }
    }
}

```





Two years in the making, *Essential ActionScript 3.0* is Colin Moock's complete guide to ActionScript 3.0. It covers programming fundamentals in truly exhaustive detail, with extreme clarity and precision. Its unparalleled accuracy and depth is the result of an entire decade of daily ActionScript research, realworld programming experience, and unmitigated insider-access to Adobe's engineers. Every word of *Essential ActionScript 3* has been carefully reviewed—in many cases several times over—by key members of Adobe's engineering staff, including those on the Flash Player, Flex Builder, and Flash authoring teams.

If you already have existing ActionScript experience, *Essential ActionScript 3.0* will help you fill in gaps in your knowledge, rethink important concepts in plain terms, and understand difficult subjects through plain, careful language. Consider *Essential ActionScript 3.0* an ActionScript expert that sits with you at your desk. You might ask it to explain the subtleties of ActionScript's event architecture, or unravel the intricacies of Flash Player's security system, or demonstrate the power of ActionScript's native XML support (E4X). Or you might turn to *Essential ActionScript 3.0* for information on under-documented topics, such as namespaces, embedded fonts, loaded-content access, class-library distribution, garbage collection, and screen updates.

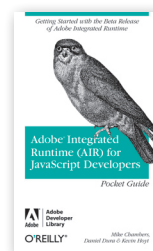
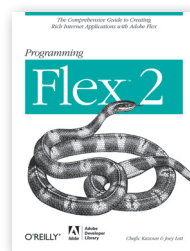
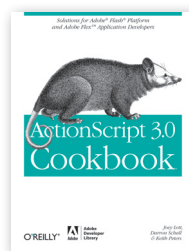
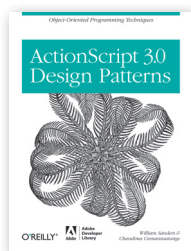
*Essential ActionScript 3.0* is a true developer's handbook, packed with practical explanations, insightful warnings, and useful example code that demonstrates how to get the job done right.

## ADOBE DEVELOPER LIBRARY



**Adobe Developer Library**, a co-publishing partnership between O'Reilly Media and Adobe Systems, Inc., is the authoritative resource for developers using Adobe technologies. With top-quality books and innovative online resources, the Adobe Developer Library delivers expert training straight from the source. Topics include ActionScript™, Adobe Flex™, Adobe Flash®, and Adobe Apollo® software.

Get the latest news about books, online resources, and more at [adobedeveloperlibrary.com](http://adobedeveloperlibrary.com)



O'REILLY



Sponsored by Adobe  
Produced by FITC.  
Handouts and books provided by O'Reilly.