Advanced ActionScript 3

Design Patterns

Master the art of object-oriented programming in ActionScript

Ben Smith
For your convenience Apress has placed some of the front matter material after the index. Please use the Bookmarks and Contents at a Glance links to access them.
Contents at a Glance

About the Author .................................................................................................................... xv
About the Technical Reviewers ........................................................................................... xvii
Acknowledgments .................................................................................................................. xix
Introduction ............................................................................................................................ xxi

■ Chapter 1: Object-Oriented Programming ........................................................................ 1
■ Chapter 2: ActionScript 3: The Facts Behind the Basics .................................................... 25
■ Chapter 3: Decision-Making and Planning ....................................................................... 45
■ Chapter 4: Intro to Design Patterns .................................................................................... 63
■ Chapter 5: Q&A .................................................................................................................. 71
■ Chapter 6: Creational Patterns ......................................................................................... 109
■ Chapter 7: Behavioral Patterns ......................................................................................... 165
■ Chapter 8: Structural Patterns ......................................................................................... 233
■ Chapter 9: Q&A .................................................................................................................. 271
■ Chapter 10: MVC: A Compound Pattern ......................................................................... 343
■ Chapter 11: Object-Oriented Design . . A Revisit ............................................................. 357
■ Chapter 12: Getting Real .................................................................................................. 373

Index .................................................................................................................................. 379
Design patterns are an abstract concept and a subject that involves being vague to help solve problems. This is somewhat ambiguous and makes design patterns a difficult topic. Fortunately, a difficult subject does not necessarily mean one that is complicated in its understanding. This will be evident in Advanced ActionScript 3: Design Patterns.

This book requires prerequisite knowledge of ActionScript and Object Oriented Programming, but it demonstrates the hand-in-hand relationship of OOP and design patterns. The beginning chapters of this book discuss and detail OOP principles, and while some aspects may be review, all will be preparation for upcoming chapters. Each chapter will prepare you for the next. Until Chapter 5 (the first review quiz), you will be reinforcing your knowledge up to that point, as well as creating a foundation for your understanding of the design pattern chapters. Chapters 6-8 thoroughly cover design patterns. Each pattern discussed is demonstrated and explained with examples, real-life analogies, and answers to frequently asked questions. Chapter 9 (the second review quiz of the book) again reinforces your knowledge up to that point. Chapters 10-12 round out the book by covering the use of combining patterns and discuss how to remain object-oriented in a fast-paced industry.

Welcome to Advanced ActionScript 3: Design Patterns.
Object-Oriented Programming

Object-oriented programming (OOP) is the practice of creating a software architecture that enables flexibility through modular design. A programmer who is object-oriented isn’t necessarily one who is a more advanced coder, but one who chooses to be a more strategic coder, and who adheres to the principles of OOP. OOP isn’t a language; it’s the practice of architecting and the thought process behind it that leads to applications and languages being object-oriented, such as ActionScript 3 (AS3).

AS3 was built as an object-oriented language to mirror the mental model of a programmer who knows the benefits of breaking code into a series of objects that can message one another. But many who choose to develop with AS3 don’t use OOP. This is due to the somewhat daunting nature of OOP, as well as the time required to learn it. AS3 is meant to support the development of flexible architecture, and using OOP can help prevent unmanageable code. Flexible architecture is easier to modify because the objects that make up the application possess distinct boundaries, which simplifies substituting among the objects you’re working with. Therefore, it’s beneficial to code with an object-oriented thought process. However, that isn’t saying you can’t use AS3 with a procedural programming mindset and be successful.

Procedural programming, which is a linear method of developing, often culminates in lines of code that have no separation of behaviors or train of thought. The language becomes nothing more than a series of routines and subroutines. Procedural programming can work well if you’re the sole developer on a project, because you’re familiar with your code. However, when more programmers are involved, it can be cumbersome for them to become familiar with one another’s code and sift through the lines to see where a change needs to be made. With OOP, each behavior in the application is contained in a unique class, providing a more elegant way to view object collaborations. Because each unique class possesses a name, it’s easy to track down; and because it should possess a single behavior, the class has only one reason to ever change.

The image in Figure 1-1 is the result of the procedural code provided in Listing 1-1. The code uses an image of my cat (Buttercup), and analyzes the pixel information to generate a halftone image.
Listing 1-1. The following code converts an image into that of a halftone

```actionscript
import flash.display.BitmapData;
import flash.display.Shape;
import flash.display.Sprite;
import flash.display.StageAlign;
import flash.geom.ColorTransform;
import flash.geom.Rectangle;

var img : BitmapData = new Buttercup( 1 , 1 );
var sampleSize : int = 4;
var brushSize : int = 4;
var pixelsTall : uint = img.height;
var pixelsWide : uint = img.width;
var rect : Rectangle = new Rectangle( 0 , 0 , sampleSize , sampleSize );
var totalBytesToScan : uint = pixelsWide * pixelsTall;
var position : uint = 0;
var offset : Number = sampleSize * 0.5;
var averageColor : uint;
var pixels : Vector.<uint>;
var darks : Number;
var halftone : Shape = new Shape();
var scale : Number;

while ( position <= totalBytesToScan )
{
    pixels = img.getVector( rect );
    averageColor = grayScaleAverage( pixels );
    darks = brightness( averageColor );
}
```

**Figure 1-1.** A color image of my cat Buttercup being converted to that of a halftone image
if ( darks > 0 )
{
    halftone.graphics.beginFill( averageColor, 1 );
    scale = (255 - darks) / 255;
    halftone.graphics.drawCircle( rect.x + offset, rect.y + offset, scale * brushSize );
}

if (rect.x >= pixelsWide)
{
    rect.x = 0;
    rect.y += sampleSize;
}
else
{
    rect.x += sampleSize;
}
position += sampleSize * sampleSize;
}

addChild( halftone );

function brightness( color : uint ) : int
{
    var R : uint = color >>16 & 0xff;
    var G : uint = color >>8 & 0xff;
    var B : uint = color & 0xff;
    return int( 0.2126 * R + 0.7152 * G + 0.0722 * B );
}

function rgbAverage( pixels : Vector.<uint> ) : uint
{
    var color : uint;
    var pixelLength : int = pixels.length;
    var averageR : uint = 0;
    var averageG : uint = 0;
    var averageB : uint = 0;
    while ( --pixelLength >=0 )
    {
        color = pixels[pixelLength];
        averageR += color >>16 & 0xFF;
        averageG += color >>8 & 0xFF;
        averageB += color & 0xFF;
    }

    averageR /=pixels.length;
    averageG /=pixels.length;
    averageB /=pixels.length;
    color = averageR << 16 | averageG << 8 | averageB;
    return color;
}
function grayScaleAverage( pixels : Vector.<uint> ) : uint
{
    var color : uint;
    var pixelLength : int = pixels.length;
    var averageR : uint;
    var averageG : uint;
    var averageB : uint;
    while ( --pixelLength >=0 )
    {
        color = pixels[pixelLength];
        averageR += color >>16 & 0xFF;
        averageG += color >>8 & 0xFF;
        averageB += color & 0xFF;
    }
    averageR /=pixels.length;
    averageG /=pixels.length;
    averageB /=pixels.length;
    var luma : int = averageR * 0.3 + averageG * 0.59 + averageB * 0.11;
    color = luma << 16 | luma << 8 | luma;
    return color;
}

This can be considered, and very well may be, a perfectly working system with only 85 lines of code. However, the code could easily begin to grow unmanageable. I even added a bit of extra code, in case I want to make a change to the system: the rgbAverage method lets me generate colored halftones if I wish.

Briefly glancing at Listing 1-1 shows it to be cumbersome and gives you little understanding about the application and how the code functions. You would probably need to analyze the code line by line to gain true insight into how the application works. But the code can be made much more organized and flexible if it’s built with the four principles of OOP in mind, encapsulation, polymorphism, inheritance, and data hiding.

Encapsulation

If you have to ask, “What am I looking at?” there is a good chance that what you’re viewing is far from the norm. For example, you know there is an engine under the hood of a car, yet you ignore such mechanics and focus on what you’re required to interact with while driving the vehicle—or so it appears. In reality, you’re concerned with what it takes to get you comfortably from point A to point B. This is known as a problem domain: what requires the focus in this case is how to remain comfortable or navigate directions.

If you’re attempting to understand engines but they don’t relate to your occupation or a hobby, you’ve probably changed your focus to a new problem domain: your broken-down engine and how you can fix it.

What you need to know per problem domain must be properly separated from what you don’t need to know, so you aren’t overloaded with extraneous information. This way, you can maintain your focus.

With this in mind, let’s apply this understanding to the halftone application. The goal of the application is to take an image and digitally alter its tone, revealing a halftone effect. If, much like the car example, you separate the engine from everything else to reveal what physically allows the application to move, then you focus on the code in Listing 1-2.
**Listing 1-2.** The “engine” of the application

import flash.display.BitmapData;
import flash.display.Shape;
import flash.display.Sprite;
import flash.display.StageAlign;
import flash.geom.ColorTransform;
import flash.geom.Rectangle;

var img : BitmapData = new Buttercup(1,1);
var sampleSize : int = 4;
var brushSize : int = 4;
var pixelsTall : uint = img.height;
var pixelsWide : uint = img.width;
var rect : Rectangle = new Rectangle(0,0,sampleSize,sampleSize);
var totalBytesToScan : uint = pixelsWide * pixelsTall;
var position : uint = 0;
var offset : Number = sampleSize * 0.5;
var averageColor : uint;
var pixels : Vector.<uint>;
var darks : Number;
var halftone : Shape = new Shape();
var scale : Number;

while ( position <= totalBytesToScan )
{
    pixels = img.getVector(rect);
    averageColor = grayScaleAverage(pixels);
    darks = brightness(averageColor);
    if ( darks > 0 )
    {
        halftone.graphics.beginFill(averageColor,1);
        scale = (255 - darks) / 255;
        halftone.graphics.drawCircle(rect.x + offset, rect.y + offset, scale * brushSize);
    }
    if (rect.x >= pixelsWide)
    {
        rect.x = 0;
        rect.y += sampleSize;
    }
    else
    {
        rect.x += sampleSize;
    }
    position += sampleSize * sampleSize;
}

addChild(halftone);

Remarkably, you’ve reduced 85 lines of code into 40 lines that define your system; this code is what you expect to find as the engine under the hood. The removed lines of code perform behaviors to be used by the engine and should be separated from the code in Listing 1-2 so you can begin to create distinguishable roles.
Polymorphism

Defining boundaries among your system's roles allows for interchangeability among other behaviors with similar method parameters, method name, and return type. These three components of a method are the contracts that proper messaging requires and together are referred to as a signature. As long as the signatures between varied implementations remain the same, the behaviors can be swapped to achieve various results without having to modify much code, if any.

Let’s consider two behaviors extracted from Listing 1-2: grayscaleAverage and rgbAverage. These behaviors are responsible for determining the average brightness of the parameterized vector of pixels and returning the calculated value. Whether the value returned possesses three color channels or one is determined by the method used to perform the calculations.

Because these two behaviors possess individual method names, the invoker of the behavior must be aware of what behavior is being called, which lessens the flexibility between the messenger and receiver.

To allow the two behaviors to be interchanged indistinguishably, you must ensure that both methods expose a common interface. Because the signatures and return types of both methods are exact, you must devise a common name by which you can invoke the method. Both methods average the brightness of a given pixel sample, so you can state that average is the common link between your algorithms (see Figure 1-2).

```
//color halftone behavior
function average(pixels:Vector.<uint> ):uint {
    // return RGB average
}

//grayscale halftone behavior
function average(pixels:Vector.<uint> ):uint {
    // return Grayscale average
}
```

*Figure 1-2. Both methods must reflect a consistent interface*

But a common interface isn’t enough to enable code substitution with procedural programming. While both methods make use of the same name, they can’t be added into the application and be compiled without throwing an error. What we need is a way to distinguish both implementations, while still making use of the common method name. Enter inheritance.

Inheritance

The concept of inheritance is modeled in object-oriented languages, enabling developers to write code in the form of hierarchical relationships. As facilitators of OOP, programmers can encapsulate a collection of behaviors and attributes into an isolated body known as an object. Such an object can then be used when you create additional objects, which you can do by deriving them from the original object. Much like children who benefit from the possessions of their mother and father, so can objects benefit through inheritance. Compartmentalized attributes and behaviors are used by child objects, creating a hierarchy between the two. A child object in a hierarchy of objects is referred to as a subclass, and its parent is referred to as its superclass.

Just as humans can be classified as mammals, any subclass in an object-oriented language can be generalized as a particular collection of attributes and behaviors of any of its ancestors. The referral to all encapsulated behaviors and attributes as objects indicates that the hierarch of all relationships is an encapsulation known as Object. Such generalization among varied implementations is required to fulfill polymorphic behavior.

To use polymorphic behaviors, your references must be typed to a generalization, thus ensuring that any and all objects to which the reference is assigned possess similar interfaces. This is so the substitution among objects doesn’t break the messaging between client and receiver.
To create a generic type that ensures both grayscale and color halftone behaviors expose the average interface, you must create a hierarchical relationship. In this case, the two behaviors are siblings that inherit the average interface from a common superclass. Not only does inheritance establish a hierarchy to allow this application to use polymorphism, but it also enables code reuse, which can minimize duplicate and repetitive code.

As Table 1-1 shows, the two implementations appear nearly identical in a side-by-side sibling comparison. The only difference between the two is the declaration of the luma variable and its calculation in the grayscale implementation.

Referring back to the concept of encapsulation, you can maintain a localized area of focus and minimize additional lines of code by appropriately situating all common code in the originator of the behavior to which the code applies. You begin by extracting variables that are common to both methods and inserting them as attributes of their superclass, as shown in Table 1-2.
### Table 1-2. Common variables are extracted from both siblings and inserted as attributes of their generic superclass

```plaintext
//generic attributes
var color : uint;
var pixelLength : int = pixels.length;
var averageR : uint;
var averageG : uint;
var averageB : uint;
var localPixels : Vector.<uint> = pixels;

//generic operation
function average( pixels : Vector.<uint> ) : uint{
    //do nothing
}
```

```plaintext
//color halftone behavior
function average(pixels:Vector.<uint>) : uint{
pixelLength = pixels.length;
localPixels = pixels;

while ( --pixelLength >=0 )
{
    color = localPixels[pixelLength];
    averageR += color >>16 & 0xFF;
    averageG += color >>8 & 0xFF;
    averageB += color & 0xFF;
}

averageR /=pixels.length;
averageG /=pixels.length;
averageB /=pixels.length;

color= averageR << 16 | averageG << 8 | averageB;
return color;
}
```

```plaintext
//grayscale halftone behavior
function average(pixels:Vector.<uint>) : uint{
pixelLength = pixels.length;
localPixels = pixels;

while ( --pixelLength >=0 )
{
    color = localPixels[pixelLength];
    averageR += color >>16 & 0xFF;
    averageG += color >>8 & 0xFF;
    averageB += color & 0xFF;
}

averageR /=pixels.length;
averageG /=pixels.length;
averageB /=pixels.length;

var luma : int = averageR * 0.3 +
    averageG * 0.59 + averageB * 0.11;

color = luma << 16 | luma << 8 | luma;
return color;
}
```

Without the excess variables, you can immediately see that minus two lines of code, both implementations are exactly the same. We can also move the averaging of all channels from both halftone algorithms into your superclass as the default implementation of the average interface; then both algorithms can use it.
Table 1-3 uses implementation inheritance, where the default implementation of the interface average in the superclass is available to both subclasses. In addition, both subclasses can redefine such inherited implementations, as shown in the table.

Table 1-3. The channel averaging among a sampled region of pixels has been localized to the superclass

```javascript
//ChannelAveraging algorithm
//generic attributes
var color : uint;
var pixelLength : int = pixels.length;
var averageR : uint;
var averageG : uint;
var averageB : uint;
var localPixels : Vector.<uint> = pixels;

//default operation
function average( pixels : Vector.<uint> ) : uint{
while ( --pixelLength >=0 )
{
    color = localPixels[pixelLength];
    averageR += color >>16 & 0xFF;
    averageG += color >>8 & 0xFF;
    averageB += color & 0xFF;
}

    averageR /=pixels.length;
    averageG /=pixels.length;
    averageB /=pixels.length;

    return null;
}

//color halftone algorithm
function average(pixels:Vector.<uint> ) : uint{
super.average(pixels);
    color = averageR << 16 | averageG << 8 | averageB;

    return color;
}

//grayscale halftone algorithm
function average(pixels:Vector.<uint> ) : uint{
    super.average(pixels);
    var luma : int = averageR * 0.3 + averageG * 0.59 + averageB * 0.11;

    color = luma << 16 | luma << 8 | luma;
    return color;
}
```
Both subclasses inherit the average implementation to which they immediately refer via the keyword `super`, which refers to the superclass's implementation of the defined method name—in this case, `average`. From there, the superclass determines the averaged channels, which are used by the remaining implementation of both algorithms. The end result is the reduction of duplicate code, the localization of logic specific to each behavior, and the increased cohesion of all three objects. We’ve also devised a generalized type where your reference can be strongly typed, enabling polymorphism between the two behaviors.

Data Hiding

Data hiding is the act of concealing information from a possible client of the application and a possible problem domain. In object-oriented languages, data hiding helps maintain proper encapsulation and is enforced by the use of namespaces such as the following:

- Attributes and behaviors that use the `private` declaration can be targeted/referenced only in the scope to which they’re declared.
- Protected is a slightly less restrictive use of `private`. Behaviors and attributes declared as protected can only be used within the class that defined them, or by that classes subclasses.
- If a class's attribute or behavior is declared as `internal`, it can be viewed by any class in the same package. By default, behaviors and attributes are always internal unless declared otherwise.
- Any attribute or behavior declared as `public` can be viewed by any class of any package.

To illustrate why data hiding is so important in OOP, refer back to Table 1-3, which shows distinct behaviors encapsulated in three unique objects. The first object calculates the average color per color channel of a sampled range of pixels. The second object calculates those channels into a hexadecimal color, which is returned to the messaging object. The third object calculates the calculated channels of the first object into a grayscale tone value, which is returned to the messaging client.

Each object has an obvious role in the application, and when a change is required or a bug occurs, the object you must modify is apparent. The code is so clear because each object maintains control over the manipulation of its own attributes—for now, at least. But when another object erroneously references a variable that doesn’t pertain to it, tracking down an error may become puzzling and delay immediate repair. Data hiding can help prevent such errors from taking place by ensuring proper visibility among messaging objects.

As shown in Table 1-4, the `average` interface is declared `public`. The attributes declared by the superclass of the halftone algorithms are another story: they’re marked as both `private` and `protected`, thus ensuring that only appropriate objects can view/manipulate such data.
You can further ensure that the attributes of the superclass are read-only to each subclassed behavior by adding public getter methods as additional interfaces of the superclass. Doing so lets each subclass retrieve attribute values without being able to reassign a value to the reference. To enforce that each subclass uses the getter methods versus reading the properties to which they currently have access, you continue to mark all protected attributes of the superclass as private.
This example illustrates the potential power of an object-oriented language. Remaining object-oriented as you write code, which makes the code easier to maintain and more flexible. Now that we’ve covered the principles of OOP, let’s focus on their implementation into an object-oriented language.

Note: It’s always easier to say how to properly engineer a better structure after all is said and done. Don’t be discouraged if you understood the previous example but can’t yet create OOP code on your own. The goal is to understand how the building blocks add modularity and flexibility while reducing the possibility of disaster, by following the four principles of OOP.

**ActionScript as an Object-Oriented Language**

Working with an object-oriented mentality opens the door to a new manner of programming. ActionScript lets you flexibly develop rich Internet applications (RIAs) when you program according to the four OOP principles:

- **Encapsulation**: ActionScript allows for the compartmentalization of behaviors and data into a class.
- **Polymorphism**: Objects within a hierarchy can respond to the operations defined by their hierarch, when indistinguishably messaged by the client.
- **Inheritance**: Like every class in the API, a custom class is an extension of the most generalized class in the language. This most basic class is appropriately called Object. The Object class makes it possible to add custom classes to a system, as long as those classes use the proper language structure and syntax.
- **Data hiding**: A class ensures its own behavioral and data security by using namespaces. In the ActionScript language, five namespace modifiers provide varying levels of security.
  - *public*: Add the keyword public in lowercase before the declaration of variables or methods. This namespace modifier provides no security. Behaviors and variables can be seen and manipulated by all classes and objects.
  - *internal*: The default namespace. This is the first tier of security in that the class is public, but only to other classes in the same package.
  - *private*: The opposite of public, allowing no access except by the class that made the private declaration.
  - *protected*: Similar to private, but visibility among properties and behaviors are available to classes which subclass the class which defines any protected attribute or behavior.
  - *final*: Ensures that either class or method cannot be extended, and thus protects all declared behaviors or classes form being modified via inheritance.
  - *Custom namespace*: Declaring a custom namespace for either a behavior or an attribute treats any such modified elements as being private, although it’s only private to classes which have not opened the custom namespace (we’ll learn more about this later in the chapter).

**Defining an External Definition**

Up to now, you’ve explored the principles of OOP and seen how the four principles of OOP work harmoniously to improve your code architecture. All that remains is to learn how to physically construct these objects in the AS3 language.
Part of the burden of defining a custom object is that such an object isn’t natively understood by the compiler. You must tell the compiler what the object does and also where to find it. Essentially, the compiler must be made aware of an external definition.

Any spoken language can have multiple definitions for a particular word, which may create confusion during a conversation. Similarly, computers require a way to tell which definition should be used; otherwise, unexpected errors may arise.

Because it’s impossible to have two files with exactly the same name and extension in the same system folder, a definition can be differentiated by its location along with its file name. Therefore, each definition can be viewed as being unique to the compiler.

The location of a definition, noted by its folder structure, becomes the pointer to the appropriate definition. This is known as a Unified Resource Identifier (URI); it enables the compiler to differentiate among definitions of similar names.

The first step in creating an external definition is writing the wrapper that surrounds the body of the definition. This is demonstrated in Table 1-5.

Table 1-5. The skeletal structure of an external definition

```java
1 package [folder[, folder... ]]
2 {
3  [visible] type_of_definition DefinitionsName
4  {
5     //BODY OF THE DEFINITION;
6  }
7 }
```

As you can see, to begin a definition, you identify its location using the package directive. This represents the folder structure where the definition resides. The arrangement of folders for the project is entirely up to you, but it’s beneficial to group your objects in a manner that represents the relationships between definitions. This grouping may be apparent in the physical naming of the folders as well as in their hierarchy. It not only keeps your classes organized, but also gives other developers a clear idea which definitions are being used and what other objects may be used with them.

As denoted by line 1 in Table 1-5 the package directive is followed by the full path to where the definition resides, denoted with dot notation. If the class doesn’t reside in a folder structure, then the folder arguments following the package directive may be left blank.

An example you’re sure to have seen is the import code for the MovieClip class (see Figure 1-3).

```java
import flash.display.MovieClip;
```

Figure 1-3. The MovieClip import reflects the MovieClip package
As you can see, MovieClip is a class in a series of nested folders that reveals the nature of the class—you don’t even need to see the code that MovieClip contains. The import refers to where the location of the definition used.

If you open the MovieClip.as file, the first line looks like: package flash.display{

Now that you’ve specified the location of your definition, ActionScript expects the type of definition and the name of the definition. The name must also be that of the saved .as file. Although the package and filename specify a particular definition, the actual definition, which varies depending on the type of definition being created, is placed in the body as shown on line 5 of Table 1-5.

ActionScript uses three types of definitions: class, namespace, and interface. The next section, describes the parts of each definition.

Parts of a Class

Even the most introductory ActionScript books cover the creation of custom classes (see Table 1-5). Often, this is to demonstrate the use of the language. It would be beyond the scope of those books to ensure that you have the proper object-oriented mindset as you construct your classes. With this in mind, don’t be too anxious to skip ahead if you feel you may already be familiar with constructing classes.

On line 4 in Figure 1-4, you used the keyword class to signify that the following definition relates to a class. The class directive can include four additional modifiers, not including its visibility. By default, you don’t see any of the keywords other than class in your editor, because everything in brackets in Figure 1-4 is optional; these elements let you make custom modifications to the class.

The first optional modifier is the keyword dynamic. Specifying that your class is dynamic lets you add properties to your class definition at runtime. Without this modifier, the class is locked: you can’t add new properties or behaviors later. Dynamic can only be used to modify the definition of the class, not the properties or methods. Subclasses of a dynamic class can’t inherit dynamic behavior, because it’s specific to the current definition.

The next modifier specifies the visibility of the class’s definition and defaults to internal, but can be specified as public to allow any classes outside the declared package view its definition. Although there are five namespaces that modify visibility, only internal and public can be used to modify the visibility of a definition.

The last optional attribute, final, specifies that the class can’t be subclassed. Because inheritance is a significant part of OOP, the use of this final keyword may be confusing, but it enforces data hiding. Declaring a definition as final prohibits any classes from subclassing the definition. This ensures that the class can’t be modified, short of physically changing the code in the original file.

Following the class directive, you add a name to identify the definition. To distinguish this from methods and variables that use camelCase, class names use a capital letter at the start of each word. Class names should be specific to the behavior they define. An appropriately named class can allude to the behaviors that a developer expects to find within the definition.

The remaining keywords (extends and implements), again optional, let you add your class to an existing hierarchy. By default, all classes extend the top-level Object unless specified otherwise. This is why it’s said that to initialize a class is to instantiate an object. At the core of every class is an Object.
Through the principle of inheritance, your class gains all public and protected, properties and behaviors of each class in the hierarchy of the chosen superclass.

Suppose we were devising a class named Foo, and Foo requires the abilities possessed by MovieClip. Choosing to subclass MovieClip looks like the following:

```java
package {
    public class Foo extends MovieClip {
        public function Foo() {
            // constructor code
        }
    }
}
```

Because ActionScript doesn’t support inheriting from multiple classes, the keyword implements allows you to add into your class the interface utilized by a specific type. By implementing an interface, in addition to adding existing public methods to your definition, you are also adding the interfaces type to the hierarchical chain of your definition as well.

Currently, our Foo class that extends MovieClip can be typed as Object, EventDispatcher, InteractiveObject, DisplayObjectContainer, Sprite, and finally MovieClip. This is because MovieClip is the subclass of another subclass, of another subclass, all the way up to the most generic class of all, Object:

```
MovieClip ➤ Sprite ➤ DisplayObjectContainer ➤ InteractiveObject ➤ DisplayObject ➤ EventDispatcher ➤ Object
```

Before you go any further, let’s see what it means to extend from EventDispatcher. As you can see in Table 1-6, EventDispatcher is a subclass of Object and implements an interface named IEventDispatcher.

<table>
<thead>
<tr>
<th>Implements</th>
<th>IEventDispatcher</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inheritance</td>
<td>EventDispatcher ➤ Object</td>
</tr>
</tbody>
</table>

IEventDispatcher is referred to as an interface, which, as the word implies, declares the public methods, and only public methods, with which you can “interface” or interact. If these methods were defined as anything other than public, then technically you couldn’t use them or interface with them—hence the term. Therefore, they must be public. An interface isn’t a class, but it requires a name to which a collection of defined methods can be referred; it too can be used to add another type to your class.

The inclusion of IEventDispatcher demonstrates that the interface is what allows MovieClips and Sprites to exhibit the public methods listed in Figure 1-5.
Now that you have your class's definition, you can add the attributes and behaviors to the definition's body. You can specify their visibility as private, public, protected, internal, or using custom namespaces, and you can also declare them as being static.

Static isn’t a visibility modifier, but it establishes whether the attribute or behavior is a member of the class or the instantiation itself. The difference is that if a behavior or attribute is an instance member, any assignment is localized to the individual object; but a class member signified via the keyword static is referenced by every instance (see Listing 1-3).

**Listing 1-3.** Demonstrates how class members are referenced by every instance

```typescript
package {
    public class StaticExample {
        public static var classString:String = 'I am a variable of the Class itself';

        public var instanceString:String = 'I am a variable of the instance';

        public function changeClassMember(str:String):void {
            classString = str;
        }

        public function changeInstanceMember(str:String):void {
            instanceString = str;
        }

        public function traceClassString():void {
            trace(classString);
        }

        public function traceInstanceString():void {
            trace(instanceString);
        }
    }
}
```

**Figure 1-5.** The IEventDispatcher interface inherited by MovieClip and Sprite