

## CHAPTER 2

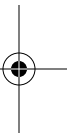
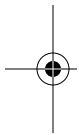
# Object-Oriented ActionScript

Ironically, Flash users who are new to object-oriented programming (OOP) are often familiar with many object-oriented concepts without knowing their formal names. This chapter demystifies some of the terminology and brings newer programmers up to speed on key OOP concepts. It also serves as a high-level overview of OOP in Flash for experienced programmers who are making their first foray into Flash development.

## Procedural Programming and Object-Oriented Programming

Traditional programming consists of various instructions grouped into *procedures*. Procedures perform some task without any knowledge of or concern for the larger program. For example, a procedure might perform a calculation and return the result. In a procedural-style Flash program, repeated tasks are stored in *functions* and data is stored in *variables*. The program runs by executing functions and changing variable values, typically for the purpose of handling input and generating output. Procedural programming is sensible for certain applications; however, as applications become larger or more complex and the interactions between procedures (and the programmers who use them) become more numerous, procedural programs can become unwieldy. They can be hard to maintain, hard to debug, and hard to upgrade.

*Object-oriented programming* (OOP) is a different approach to programming, intended to solve some of the development and maintenance problems commonly associated with large procedural programs. OOP is designed to make complex applications more manageable by breaking them down into self-contained, interacting modules. OOP lets us translate abstract concepts and tangible real-world things into corresponding parts of a program (the “objects” of OOP). It’s also designed to let an application create and manage more than one of something, as is often required by



user interfaces. For example, we might need 20 cars in a simulation, 2 players in a game, or 4 checkboxes in a fill-in form.

Properly applied, OOP adds a level of conceptual organization to a program. It groups related functions and variables together into separate *classes*, each of which is a self-contained part of the program with its own responsibilities. Classes are used to create individual objects that execute functions and set variables on one another, producing the program's behavior. Organizing the code into classes makes it easier to create a program that maps well to real-world problems with real-world components. Parts II and III of this book cover some of the common situations you'll encounter in ActionScript and show how to apply OOP solutions to them. But before we explore applied situations, let's briefly consider the basic concepts of OOP.

## Key Object-Oriented Programming Concepts

An *object* is a self-contained software module that contains related functions (called its *methods*) and variables (called its *properties*). Individual objects are created from *classes*, which provide the blueprint for an object's methods and properties. That is, a class is the template from which an object is made. Classes can represent theoretical concepts, such as a timer, or physical entities in a program, such as a pull-down menu or a spaceship. A single class can be used to generate any number of objects, each with the same general structure, somewhat as a single recipe can be used to bake any number of muffins. For example, an OOP space fighting game might have 20 individual *SpaceShip* objects on screen at one time, all created from a single *SpaceShip* class. Similarly, the game might have one *2dVector* class that represents a mathematical vector but thousands of *2dVector* objects in the game.



The term *instance* is often used as a synonym for *object*. For example, the phrases “Make a new *SpaceShip* instance” and “Make a new *SpaceShip* object” mean the same thing. Creating a new object from a class is sometimes called *instantiation*.

To build an object-oriented program, we:

1. Create one or more classes.
2. Make (i.e., *instantiate*) objects from those classes.
3. Tell the objects what to do.

What the objects *do* determines the behavior of the program.

In addition to using the classes we create, a program can use any of the classes built into the Flash Player. For example, a program can use the built-in *Sound* class to create *Sound* objects. An individual *Sound* object represents and controls a single sound or a group of sounds. Its *setVolume()* method can raise or lower the volume of a

sound. Its `loadSound()` method can retrieve and play an MP3 sound file. And its `duration` property can tell us the length of the loaded sound, in milliseconds. Together, the built-in classes and our custom classes form the basic building blocks of all OOP applications in Flash.

## Class Syntax

Let's jump right into a tangible example. Earlier, we suggested that a space fighting game would have a `SpaceShip` class. The ActionScript that defines the class might look like the source code shown in Example 2-1 (don't worry if much of this code is new to you; we'll study it in detail in the coming chapters).

*Example 2-1. The SpaceShip class*

```
class SpaceShip {
    // This is a public property called speed.
    public var speed:Number;

    // This is a private property called damage.
    private var damage:Number;

    // This is a constructor function, which initializes
    // each SpaceShip instance.
    public function SpaceShip () {
        speed = 100;
        damage = 0;
    }

    // This is a public method called fireMissile().
    public function fireMissile ():Void {
        // Code that fires a missile goes here.
    }

    // This is a public method called thrust().
    public function thrust ():Void {
        // Code that propels the ship goes here.
    }
}
```

Notice how the `SpaceShip` class groups related aspects of the program neatly together (as do all classes). Variables (properties), such as `speed` and `damage`, related to spaceships are grouped with functions (methods) used to move a spaceship and fire its weapons. Other aspects of the program, such as keeping score and drawing the background graphics can be kept separate, in their own classes (not shown in this example).

## Object Creation

Objects are created (instantiated) with the `new` operator, as in:

```
new ClassName()
```

where *ClassName* is the name of the class from which the object will be created.

For example, when we want to create a new *SpaceShip* object in our hypothetical game, we use this code:

```
new SpaceShip()  
    AS1 note
```

The syntax for creating objects (e.g., *new SpaceShip()*) is the same in ActionScript 2.0 as it was in ActionScript 1.0. However, the syntax for defining classes in ActionScript 2.0 differs from ActionScript 1.0.

Most objects are stored somewhere after they're created so that they can be used later in the program. For example, we might store a *SpaceShip* instance in a variable named *ship*, like this:

```
var ship:SpaceShip = new SpaceShip();
```

Each object is a discrete data value that can be stored in a variable, an array element, or even a property of another object. For example, if you create 20 alien spaceships, you would ordinarily store references to the 20 *SpaceShip* objects in a single array. This allows you to easily manipulate multiple objects by cycling through the array and, say, invoking a method of the *SpaceShip* class on each object.

## Object Usage

An object's methods provide its capabilities (i.e., behaviors)—things like “fire missile,” “move,” and “scroll down.” An object's properties store its data, which describes its state at any given point in time. For example, at a particular point in a game, our *ship*'s current state might be speed is 300, damage is 14.

Methods and properties that are defined as *public* by an object's class can be accessed from anywhere in a program. By contrast, methods and properties defined as *private* can be used only within the source code of the class or its subclasses. As we'll learn in Chapter 4, methods and properties should be defined as *public* only if they must be accessed externally.

To invoke a method, we use the dot operator (i.e., a period) and the function call operator (i.e., parentheses). For example:

```
// Invoke the ship object's fireMissile() method.  
ship.fireMissile();
```

To set a property, we use the dot operator and an equals sign. For example:

```
// Set the ship's speed property to 120.  
ship.speed = 120;
```

To retrieve a property's value, we use the dot operator on its own. For example:

```
// Display the value of the speed property in the Output panel.  
trace(ship.speed);
```

## Encapsulation

Objects are said to *encapsulate* their property values and method source code from the rest of the program. If properly designed, an object's private properties and the internal code used in its methods (including public methods) are its own business; they can change without necessitating changes in the rest of the program. As long as the method names (and their parameters and return values) stay the same, the rest of the program can continue to use the object without being rewritten.

Encapsulation is an important aspect of object-oriented design because it allows different programmers to work on different classes independently. As long as they agree on the names of the public methods through which they'll communicate, the classes can be developed independently. Furthermore, by developing a specification that shows the publicly available methods, the parameters they require, and the values they return, a class can be tested thoroughly before being deployed. The same test code can be used to reverify the class's operation even if the code within the class is *refactored* (i.e., rewritten to enhance performance or to simplify the source code without changing the previously existing functionality).

In Chapter 4, we'll learn how to use the *private* modifier to prevent a method or property from being accessed by other parts of a program.

## Datatypes

Each class in an object-oriented program can be thought of as defining a unique kind of data, which is formally represented as a *datatype* in the program.



A class effectively defines a custom datatype.

You are probably already familiar with custom datatypes defined by built-in ActionScript classes, such as the *Date* class. That is, when you create a *Date* object using *new Date()*, the returned value contains not a string or a number but a complex datatype that defines a particular day of a particular year. As such, the *Date* datatype supports various properties and methods uniquely associated with dates.

Datatypes are used to impose limits on what can be stored in a variable, used as a parameter, or passed as a return value. For example, when we defined the *speed* property earlier, we also specified its datatype as *Number* (as shown in bold):

```
// The expression ":Number" defines speed's datatype.  
public var speed:Number;
```

Attempts to store a nonnumeric value in the `speed` property generate a compile-time error.

If you test a movie and Flash's Output panel displays an error containing the phrase "Type mismatch," you know that you used the wrong kind of data somewhere in your program (the compiler will tell you precisely where). Datatypes help us guarantee that a program isn't used in unintended ways. For example, by specifying that the datatype of `speed` is a number, we prevent someone from unintentionally setting `speed` to, say, the string "very fast." The following code generates a compile-time error due to the datatype mismatch:

```
public var speed:Number = "very fast"; // Error!  
                                     // You can't assign a String to a  
                                     // variable whose type is Number.
```

We'll talk more about datatypes and type mismatches in Chapter 3.

## Inheritance

When developing an object-oriented application, we can use *inheritance* to allow one class to adopt the method and property definitions of another. Using inheritance, we can structure an application hierarchically so that many classes can reuse the features of a single class. For example, specific *Car*, *Boat*, and *Plane* classes could reuse the features of a generic *Vehicle* class, thus reducing redundancy in the application. Less redundancy means less code to write and test. Furthermore, it makes code easier to change—for example, updating a movement algorithm in a single class is easier and less error prone than updating it across several classes.

A class that inherits properties and methods from another class is called a *subclass*. The class from which a subclass inherits properties and methods is called the subclass's *superclass*. Naturally, a subclass can define its own properties and methods in addition to those it inherits from its superclass. A single superclass can have more than one subclass, but a single subclass can have only one superclass (although it can also inherit from its superclass's superclass, if any). We'll cover inheritance in detail in Chapter 6.

## Packages

In a large application, we can create *packages* to contain groups of classes. A package lets us organize classes into logical groups and prevents naming conflicts between classes. This is particularly useful when components and third-party class libraries are involved. For example, Flash MX 2004's GUI components, including one named *Button*, reside in a package named *mx.controls*. The GUI component class named *Button* would be confused with Flash's built-in *Button* class if it weren't identified as part of the *mx.controls* package. Physically, packages are directories that are collections of class files (i.e., collections of *.as* files).

We'll learn about preventing naming conflicts by referring to classes within a package, and much more, in Chapter 9.

## Compilation

When an OOP application is exported as a Flash movie (i.e., a *.swf* file), each class is *compiled*; that is, the compiler attempts to convert each class from source code to *bytecode*—instructions that the Flash Player can understand and execute. If a class contains errors, compilation fails and the Flash compiler displays the errors in the Output panel in the Flash authoring tool. The error messages, such as the datatype mismatch error described earlier, should help you diagnose and solve the problem. Even if the movie compiles successfully, errors may still occur while a program is running; these are called *runtime errors*. We'll learn about Player-generated runtime errors and program-generated runtime errors in Chapter 10.

## Starting an Objected-Oriented Application

In our brief overview of OOP in Flash, we've seen that an object-oriented application is made up of classes and objects. But we haven't learned how to actually start the application running. Every Flash application, no matter how many classes or external assets it contains, starts life as a single *.swf* file loaded into the Flash Player. When the Flash Player loads a new *.swf* file, it executes the actions on frame 1 and then displays the contents of frame 1.

Hence, in the simplest case, we can create an object-oriented Flash application and start it as follows:

1. Create one or more classes in *.as* files.
2. Create a *.fla* file.
3. On frame 1 of the *.fla* file, add code that creates an object of a class.
4. Optionally invoke a method on the object to start the application.
5. Export a *.swf* file from the *.fla* file.
6. Load the *.swf* file into the Flash Player.

We'll study more complex ways to structure and run object-oriented Flash applications in Chapters 5, 11, and 12.

## But How Do I Apply OOP?

Many people learn the basics of OOP only to say, "I understand the terminology and concepts, but I have no idea how or when to use them." If you have that feeling, don't worry, it's perfectly normal; in fact, it means you're ready to move on to the next phase of your learning—*object-oriented design* (OOD).

The core concepts of OOP (classes, objects, methods, properties, etc.) are only tools. The real challenge is designing what you want to build with those tools. Once you understand a hammer, nails, and wood, you still have to draw a blueprint before you can actually build a fence, a room, or a chair. Object-oriented design is the “draw a blueprint” phase of object-oriented programming, during which you organize your entire application as a series of classes. Breaking a program up into classes is a fundamental design problem that you’ll face daily in your OOP work. We’ll return to this important aspect of OOP regularly throughout this book.

But not all Flash applications need to be purely object-oriented. Flash supports both procedural and object-oriented programming and allows you to combine both approaches in a single Flash movie. In some cases, it’s sensible to apply OOP to only a single part of your project. Perhaps you’re building a web site with a section that displays photographs. You don’t have to make the whole site object-oriented; you can just use OOP to create the photograph-display module. (In fact, we’ll do just that in Chapters 5 and 7!)

So if Flash supports both procedural and object-oriented programming, how much of each is right for your project? To best answer that question, we first need to understand the basic structure of every Flash movie. The fundamental organizing structure of a Flash document (a *.fla* file) is the *timeline*, which contains one or more *frames*. Each frame defines the content that is displayed on the graphical canvas called the *Stage*. In the Flash Player, frames are displayed one at a time in a linear sequence, producing an animated effect—exactly like the frames in a filmstrip.

At one end of the development spectrum, Flash’s timeline is often used for interactive animation and motion graphics. In this development style, code is mixed directly with timeline frames and graphical content. For example, a movie might display a 25-frame animation, then stop, calculate some random feature used to display another animation, then stop again and ask the user to fill in a form while yet another animation plays in the background. That is, for simple applications, different frames in the timeline can be used to represent different program *states* (*each state is simply one of the possible places, either physical or conceptual, that a user can be in the program*). For example, one frame might represent the welcome screen, another frame might represent the data entry screen, a third frame might represent an error screen or exit screen, and so on. Of course, if the application includes animation, each program state might be represented by a range of frames instead of a single frame. For example, the welcome screen might include a looping animation.

When developing content that is heavily dependent on motion graphics, using the timeline makes sense because it allows for precise, visual control over graphic elements. In this style of development, code is commonly attached to the frames of the timeline. The code on a frame is executed immediately before the frame’s content is displayed. Code can also be attached directly to the graphical components on stage.

For example, a button can contain code that governs what happens when it is clicked.

Timeline-based development usually goes hand-in-hand with procedural programming because you want to take certain actions at the time a particular frame is reached. In Flash, “procedural programming” means executing code, defining functions, and setting variables on frames in a document’s timeline or on graphical components on stage.

However, not all Flash content necessarily involves timeline-based motion. If you are creating a video game, it becomes impossible to position the monsters and the player’s character using the timeline. Likewise, you don’t know exactly when the user is going to shoot the monster or take some other action. Therefore, you must use ActionScript instead of the timeline to position the characters in response to user actions (or in response to an algorithm that controls the monsters in some semi-intelligent way). Instead of a timeline-based project containing predetermined animated sequences, we have a nonlinear project in which characters and their behavior are represented entirely in code.

This type of development lends itself naturally to objects that represent, say, the player’s character or the monsters. At this end of the development spectrum lies traditional object-oriented programming, in which an application exists as a group of classes. In a pure object-oriented Flash application, a *.fla* file might contain a single frame only, which simply loads the application’s main class and starts the application by invoking a method on that main class. Of course, OOP is good for more than just video games. For example, a Flash-based rental car reservation system might have no timeline code whatsoever and create all user interface elements from within classes.

Most real-world Flash applications lie somewhere between the extreme poles of timeline-only development and pure OOP development. For example, consider a Flash-based web site in which two buttons slide into the center of the screen and offer the user a choice of languages: “English” or “French.” The user clicks the preferred language button, and both buttons slide off screen. An animated sequence then displays company information and a video showing a product demo. The video is controlled by a *MediaPlayback* component.

Our hypothetical web site includes both procedural programming and OOP, as follows:

- Frames 2 and 3 contain preloader code.
- Frame 10 contains code to start the button-slide animation.
- Frames 11–25 contain the button-slide animation.
- Frame 25 contains code to define button event handlers, which load a language-specific movie.

- In the loaded language-specific movie, frame 1 contains code to control the *MediaPlayback* component.

In the preceding example, code placed directly on frames (e.g., the preloader code) is procedural. But the buttons and *MediaPlayback* component are objects derived from classes stored in external *.as* files. Controlling them requires object-oriented programming. And, interestingly enough, Flash components are, themselves, movie clips. Movie clips, so intrinsic to Flash, can be thought of as self-contained objects with their own timelines and frames. Components (indeed, any movie clip) can contain procedural code internally on their own frames even though they are objects. Such is the nature of Flash development—assets containing procedural code can be mixed on multiple levels with object-oriented code.



As mentioned in the Preface, this book assumes you understand movie clips and have used them in your work. If you are a programmer coming to Flash from another language, and you need a crash course on movie clips from a programmer's perspective, consult Chapter 13 of *ActionScript for Flash MX: The Definitive Guide*, available online at: <http://moock.org/asdg/samples>.

Flash's ability to combine procedural and object-oriented code in a graphical, time-based development environment makes it uniquely flexible. That flexibility is both powerful and dangerous. On one hand, animations and interface transitions that are trivial in Flash might require hours of custom coding in languages such as C++ or Java. But on the other hand, code that is attached to frames on timelines or components on the Stage is time-consuming to find and modify. So overuse of timeline code in Flash can quickly (and quietly!) turn a project into an unmaintainable mess. Object-oriented techniques stress separation of code from assets such as graphics and sound, allowing an object-oriented application to be changed, reused, and expanded upon more easily than a comparable timeline-based program. If you find yourself in the middle of a timeline-based project faced with a change and dreading the work involved, chances are the project should have been developed with object-oriented principles from the outset. Although OOP may appear to require additional up-front development time, for most nontrivial projects, you'll reclaim that time investment many times over later in the project.

Ultimately, the amount of OOP you end up using in your work is a personal decision that will vary according to your experience and project requirements. You can use the following list to help decide when to use OOP and when to use procedural timeline code. Bear in mind, however, that these are just guidelines—there's always more than one way to create an application. Ultimately, if the software works and can be maintained, you're doing something right.

Consider using OOP when creating:

- Traditional desktop-style applications with few transitions and standardized user interfaces
- Applications that include server-side logic
- Functionality that is reused across multiple projects
- Components
- Games
- Highly customized user interfaces that include complex visual transitions

Consider using procedural programming when creating:

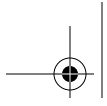
- Animations with small scripts that control flow or basic interactivity
- Simple applications such as a one-page product order form or a newsletter subscription form
- Highly customized user interfaces that include complex visual transitions

You'll notice that "Highly customized user interfaces that include complex visual transitions" is included as a case in which you might use both OOP and procedural programming. Both disciplines can effectively create that kind of content. However, remember that OOP in Flash is typically more maintainable than timeline code and is easier to integrate into version control systems and other external production tools. If you suspect that your highly customized UI will be used for more than a single project, you should strongly consider developing it as a reusable class library or set of components with OOP.

Note that in addition to Flash's traditional timeline metaphor, Flash MX Professional 2004 introduced a *Screens* feature (which includes both *Slides* and *Forms*). Screens provide a facade over the traditional timeline metaphor. Slides and Forms are akin to the card-based metaphors of programs like HyperCard. Slides are intended for PowerPoint-style slide presentations, while Forms are intended for VB developers used to working on multipage forms. Like timeline-based applications, Screens-based applications include both object-oriented code (i.e., code in classes) and procedural-style code (i.e., code placed visually on components and on the Screens of the application). As mentioned in the Preface, this book does not cover Screens in detail, but the foundational OOP skills you'll learn in this text will more than equip you for your own exploration of Screens.

## On with the Show!

In this chapter, we summarized the core concepts of OOP in Flash. We're now ready to move on with the rest of Part I, where we'll study all of those concepts again in detail, applying them to practical situations along the way. If you're already quite comfortable with OOP and want to dive into some examples, see chapters 5, 7, 11,



and 12 and all of Part III, which contain in-depth studies of real-world object-oriented code.

Let's get started!

