

# ActionScript 3.0 Cookbook

## Custom Classes

By Joey Lott, Darron Schall and Keith Peters

Publisher: O'Reilly Media, Inc.

Pub Date: 2006-10-11

ISBN: 9780596526955

---

### Table Of Contents

Custom Classes.....	3
Introduction	3
Creating a Custom Class	3
Determining Where to Save a Class	7
Creating Properties That Behave As Methods	8
Creating Static Methods and Properties	10
Creating Subclasses	11
Implementing Subclass Versions of Superclass Methods	13
Creating Constants	14
Dispatching Events	15
Index.....	17

Copyright (©) 2004, 2005, 2006, 2007 O'Reilly Media.

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472

The copyrights in individual elements of this work are owned by their respective publishers, authors or others, as the case may be, and the prior written permission of the copyright owner is required for reuse in any form or medium of any individual element.

O'Reilly books may be purchased for educational, business or sales promotional use. Online editions are also available for most titles (<http://safari.oreilly.com/>). For more information, contact our corporate/institutional sales department: (800)998.9938 or [corporate@oreilly.com](mailto:corporate@oreilly.com).

Nutshell Handbook, the Nutshell Handbook logo, and the O'Reilly logo are registered trademarks of O'Reilly Media, Inc. Microsoft, the .NET logo, Virtual C#, Visual Basic, Visual Studio, and Windows are registered trademarks or trademarks of Microsoft Corporation. Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and O'Reilly Media, Inc. was aware of a trademark claim, the designations have been printed in caps or initial caps. The association between the image of the African crowned crane and the topic of C# is a trademark of O'Reilly Media, Inc.

While reasonable care has been exercised in the preparation of this book, the publisher and the authors assume no responsibility for errors or omissions, or for damages resulting from the use of the information contained herein.

---

# Custom Classes

## Introduction

Classes are absolutely essential to ActionScript 3.0. This is truer in ActionScript 3.0 than in any earlier release of the language. ActionScript 1.0 was essentially a procedural language with modest object-oriented features. ActionScript 2.0 formalized the object-oriented features and took a big step in the direction of a truly object-oriented language. However, ActionScript 3.0 shifts the core focus of ActionScript so that the basic building block is that of the class. If you are using ActionScript 3.0 with Flex, and the introduction of the minor exception of code being placed within `<mx:Script>` tags, all ActionScript code must appear within a class. This chapter discusses the fundamentals of writing custom classes in ActionScript 3.0.

## Creating a Custom Class

### Problem

You want to write a custom class.

### Solution

Save a new file with the `.as` file extension and the filename matching the name of the class. Then add the class definition to the file with the following structure:

```
package package {
    public class Class {

    }
}
```

### Discussion

As noted earlier, the class is the basic building block of all ActionScript 3.0-based applications, so it's essential that you master the basics of writing a class. For starters, all classes must be placed in `.as` files, which are plain text files saved with an `.as` file extension. There can be only one public class definition per `.as` file, and the name of the file must be the same as the name of the class. For example, if you name a class *Example* then the definition must be saved in a file called *Example.as*.

In ActionScript 3.0 all classes must be placed in packages. A package is a way of organizing classes into groups, and in ActionScript 3.0 a package is synonymous with a directory on the filesystem. Packages are relative to the classpath (which is discussed in detail in *Recipe 2.2*), but for this initial discussion the classpath is defined as a path relative to the project (the `.fla` file, in the case of Flash or the main class or MXML document in the case of Flex). Therefore, the top-level package is

synonymous with the project's root. The package declaration is always the first thing that appears in a class file; an example syntax follows:

```
package name {  
  
}
```

When the class is defined as part of the top-level package, the package name doesn't need to be specified. In those cases, the package declaration is as follows:

```
package {  
  
}
```

When the class file is saved within a subdirectory, the package name corresponds to the relative path of the subdirectory using dots (.) between each directory. For example, if a file is saved in a subdirectory (relative to the project root) called *example*, then the package declaration is as follows:

```
package example {  
  
}
```

If the class file is saved in a subdirectory of *example* called *subpackage*, the package declaration is as follows:

```
package example.subpackage {  
  
}
```

Packages are an important part of working with classes because they allow you to ensure that your classes won't conflict with any other classes. For example, it's entirely possible that two developers might write two different classes and name them *MessageManager*. The two classes could have the same name but be responsible for very different tasks. One might manage emails while the other manages binary socket messages. You cannot have two classes with the same name in the same scope. If you did, the compiler wouldn't know which one to use. One option is to always use unique class names.

In this example, you could name the classes *EmailManager* and *BinarySocket-MessageManager*, but there are good reasons why that isn't always possible or preferable. Since a project could use potentially hundreds of classes, it would be very difficult to ensure all the classes have unique names. Furthermore, many projects may rely on preexisting libraries of code, and you may not have written many of the classes. Because many classes in a library might have dependencies on other classes, it would be very difficult to change the names of classes. This is where packages make things much simpler. Even though you cannot have two classes with the same name within the same directory, you can have as many classes with the same name as you want, so long as they are within different directories. Packages allow you to place one *MessageManager* in the *net.messaging.email* package and one in the *net.messaging.binarysocket* package.

Although the preceding package names may initially seem like good choices, it's possible to use better package names to ensure a higher probability of uniqueness. It's generally better to use package names that correspond to the owner and/or relevant project. By convention, package names start with reverse order domain names. For example, if Example Corp (*examplecorp.com*) writes ActionScript 3.0 classes, they would place all classes in the *com.examplecorp* package (or a subpackage of *com.examplecorp*). That way, if another Example Corp from the U.K.

(*examplecorp.co.uk*) also writes ActionScript 3.0 classes, they can ensure uniqueness by using the package *uk.co.examplecorp*.



The exception to this guideline is when the classes are part of a product or library that transcends company/organization boundaries. For example, all the native Flash Player classes are in the Flash package (for example, *flash.net.URLRequest*) and all the classes in the *ActionScript 3.0 Cookbook* library are in the *ascb* package (see <http://www.rightactionscript.com/ascb>).

When classes are part of a common library used by many projects within a company/organization then they can be placed in subpackages directly within the main package. For example, if the aforementioned *MessageManager* classes are part of a common library used by many Example Corp applications then they could be placed in the *com.examplecorp.net.messaging.email* and *com.examplecorp.net.messaging.binary-socket* packages. When a class is part of a specific application, it should be placed within a subpackage specific to that application. For example, Example Corp might have an application called *WidgetStore*. If the *WidgetStore* application uses a class called *ApplicationManager*, then it ought to be placed within *com.examplecorp.widgetstore* or a subpackage of that package.



By convention, package names start with lowercase characters.

The next step is to declare the class itself, as follows:

```
public class Name {  
  
}
```

The name of a class starts with a capital letter by convention. The name of a class must follow the same naming rules as variables and functions (consists of letters, numbers, and underscores, and cannot start with a number). The class declaration appears within the package declaration. The following code defines a class called *Example* within the top-level package:

```
package {  
    public class Example {  
  
    }  
}
```

The class body appears within the class declaration's curly braces, and it consists of properties and methods. Properties are variables associated with the class, and you declare them much as you would declare variables by using the *var* keyword. However, properties also must be modified by attributes, which determine the scope of the property. The following is a list of the attributes you can use with properties:

*private*

Properties are private when they're accessible only within the class.

*public*

Properties are public when they're accessible within the class as well as from instances of the class (or directly from the class reference when declared as static).

### *protected*

Properties are protected when they're accessible only within the class and to subclasses.

### *internal*

Properties are internal when they're accessible within the package.

The default state of a property is `internal` unless you specify a different attribute. In most cases, properties should be declared as either `private` or `protected`. By convention, it's helpful to start `private` and `protected` property names with an underscore (`_`). The following declares a new `private` property called `_id` within the *Example* class:

```
package {
    public class Example {
        private var _id:String;
    }
}
```

Methods are essentially functions associated with a class, and you can declare a method much like you would declare a function using the `function` keyword. However, as with properties, all methods must belong to a namespace defined by one of the attributes from the previous list. The `public`, `private`, `protected`, and `internal` attributes work identically for methods and properties. Methods should be declared `public` only when they need to be called from instances of the class (or from the class itself when declared as `static`). If the method is designed to be called only from within the class, then it should be declared as `private` or `protected`. A method should be `protected` only when you want to be able to reference it from a subclass. The following declares a method called *getId()*:

```
package {
    public class Example {
        private var _id:String;
        public function getId():String {
            return _id;
        }
    }
}
```

Method names are subject to the same rules as variables and properties. That means that method names must contain only numbers, letters, underscores, and dollar signs. Additionally, while method names can contain numbers, they cannot begin with numbers. By convention, method names start with lowercase characters. There is one exception to that guideline, however. Every class can have a special method with the same name as the class itself. This special method is called the *constructor*, and as the name implies, you can use the function to construct new instances of the class. In ActionScript 3.0, all constructors must be `public` (this is a change from ActionScript 2.0). Unlike standard methods, constructors cannot return values, and they must not declare a return type. The following declares a constructor method for the *Example* class:

```
package {
    public class Example {
        private var _id:String;
        public function Example() {
            _id = "Example Class";
        }
        public function getId():String {
            return _id;
        }
    }
}
```

```
    }  
}
```

The following illustrates how to construct an instance of the *Example* class:

```
var example:Example = new Example( );  
trace(example.getId( )); // Displays: Example Class
```

See Also

*Recipe 2.2*

## Determining Where to Save a Class

### Problem

You want to determine where to save a class file.

### Solution

Save the file in a directory path corresponding to the package. Then, if necessary, add the top-level directory to the classpath.

### Discussion

Class files must always be saved in a directory path that corresponds to the class package. For example, *com.examplecorp.net.messaging.email.MessageManager* must be saved in *com/examplecorp/net/messaging/email/MessageManager.as*. The compiler knows to look for classes where the path corresponds to the package. However, the compiler also must know where to look for the top-level directory containing the subdirectories. In the example, the compiler needs to know where the *com* directory is on the system. The compiler knows where to look because of something called the *classpath*. The default classpath for any Flex or Flash project includes the project directory. For example, if the *com* directory is saved in the same directory as the *.fla* file (Flash) or the main MXML or ActionScript file (Flex), then the compiler will find the classes. However, you may want to save files in a different directory. For example, if you have a common library used by many projects, you may want to save that library in one location rather than making copies for each project. You can add to and edit the classpath so the compiler knows where to look for all your custom classes.

For Flash, you can edit the classpath either at the project level or globally. At the project level, select **File** **Publish Settings**, and select the **ActionScript Settings** button in the **Flash** tab. For the global classpath, select **Edit** **Preferences**, and click the **ActionScript Settings** button. Both open a similar dialog that allows you to edit the classpath. You can click the **+** button to add a new directory to the classpath. For example, if the *com* directory for a common library is stored in *C:\libraries*, then you would add *C:\libraries* to the classpath.

For Flex, you can only set the classpath for a project. Using Flex Builder, right-click a project in the project navigator, select **Properties**, and then select **Flex/ActionScript Build Path** from the left menu. In the **Source Path** tab, you can add to and edit the classpath. If you're using the SDK rather than Flex Builder, you have to set the classpath when you're compiling your project. Using *mxmhc*

(the command-line compiler included in the Flex SDK), you can add a `-source-path` option, followed by a list of classpath directories, as shown here:

```
mxm1c -source-path . C:\libraries ExampleApplication.as
```

See Also

*Recipe 2.1*

## Creating Properties That Behave As Methods

### Problem

You want to use `public` properties that behave like methods so you don't break encapsulation.

### Solution

Use implicit getters and setters.

### Discussion

As mentioned in *Recipe 2.1*, all properties should be declared as `private` or `protected`. `public` properties are not a good idea because of a principal called *encapsulation*. Good encapsulation is something to strive for. It means that a class doesn't expose its internals in a way that it can be easily broken; `public` properties can enable developers to easily break a class or an instance of a class. Consider the following simple example that uses a `public` property:

```
package {
    public class Counter {
        public var count:uint;
        public function Counter() {
            count = 0;
        }
    }
}
```

You can then construct an instance of *Counter*, and you can change the `count` property value, as shown here:

```
var counter:Counter = new Counter();
counter.count++;
```

However, what if the business rules of the application state that a *Counter* should never exceed 100? You can see that the *Counter* class with a `public` `count` property makes it quite easy to break that rule.

One option is to use explicit getters and setters, as in the following example:

```
package {
    public class Counter {
        private var _count:uint;
        public function Counter() {
            _count = 0;
        }
    }
}
```

```

    public function getCount():uint {
        return _count;
    }
    public function setCount(value:uint):void {
        if(value < 100) {
            _count = value;
        }
        else {
            throw Error();
        }
    }
}

```

Another option is to use implicit getters and setters. Implicit getters and setters are declared as methods, but they look like properties. The syntax for a getter is as follows:

```

public function get name():Datatype {
}

```

The syntax for a setter is as follows:

```

public function set name(value:Datatype):void {
}

```

The following defines count with implicit getter and setter methods:

```

package {
    public class Counter {
        private var _count:uint;
        public function Counter() {
            _count = 0;
        }
        public function get count():uint {
            return _count;
        }
        public function set count(value:uint):void {
            if(value < 100) {
                _count = value;
            }
            else {
                throw Error();
            }
        }
    }
}

```

You can then treat count as though it were a public property:

```

counter.count = 5;
trace(counter.count);

```

See Also

*Recipe 2.1*

# Creating Static Methods and Properties

## Problem

You want to create methods and properties that are directly accessible from the class rather than from instances of the class.

## Solution

Use the `static` attribute when declaring the property or method.

## Discussion

By default, properties and methods are instance properties and methods, which means they are defined for each instance of the class. If the *Example* class defines a `_id` property and a `getId()` method then, by default, each instance of *Example* has its own `_id` property and `getId()` method. However, there are cases in which you want the property or method to be associated with the class itself rather than with instances of the class. That means that no matter how many instances of the class there may be, there is just one property or method. Such properties and methods are called *static properties and methods*.

There are examples of static properties and methods in several of the intrinsic Flash Player classes. For example, the *Math* class defines a `round()` method. The `round()` method is static and is, therefore, accessible directly from the class:

```
trace(Math.round(1.2345));
```

The *Math* class consists entirely of static methods and constants. However, a class can have both static and instance methods and/or properties. For example, the *String* class consists primarily of instance properties and methods. However, the `fromCharCode()` method is declared as static. The `fromCharCode()` method returns a string based on the character codes passed to the method. Since the method isn't associated with any one *String* instance, it does not make sense to make the method an instance method. However, it does make sense to declare the method as a static method.

You can declare a property or method as static using the `static` attribute. The `static` attribute is always used in combination with the `public`, `private`, `protected`, or `internal` attribute.

For example, the following declares a `private static` property called `_example`:

```
static private var _example:String;
```

The order in which the attributes appear doesn't matter. For example, `static private` is the equivalent to `private static`.

One common and important use of static properties and methods is the Singleton design pattern, whereby a class has a single managed instance. Singleton classes have a `private static` property that stores the one instance of the class as well as a `public static` method that allows access to the one instance.

## See Also

*Recipe 2.1*

# Creating Subclasses

## Problem

You want to create a class that inherits from an existing class.

## Solution

Write a subclass using the `extends` keyword.

## Discussion

There are cases when a new class is a more specific version of an existing class. The new class may feature much of the same behavior as the existing class. Rather than rewriting all the common functionality you can define the new class so it inherits all the functionality of the existing class. In relation to one another, the new class is then called a *subclass* and the existing class is called a *superclass*.

You can define inheritance between classes in the subclass declaration using the `extends` keyword, as follows:

```
public class Subclass extends Superclass
```

A subclass can reference any `public` or `protected` properties and methods of the superclass. `private` properties and methods are not accessible outside the class, not even to a subclass.

Inheritance is a powerful technique; however, as with anything else, it is important that you use inheritance correctly. Before writing a subclass you need to determine whether or not the new class actually has a subclass relationship with the existing class. There are two basic types of relationships that classes can have: inheritance and composition. You can usually quickly determine the correct relationship between classes by asking whether it's an "is a" relationship or a "has a" relationship:

- "Is a" relationships are often inheritance relationships. As an example, consider an application that manages a library's collection.
- "Has a" relationships are composition relationships in which a class declares a property. Most classes use composition. Oftentimes composition can be implemented in such a way that it achieves the same results as inheritance with greater flexibility (yet generally requiring more code). For example, a book is not an author, but it has an author (or authors).

The library has different types of items in the collection including books and DVDs. Obviously books and DVDs have different types of data associated with them. Books have page counts and authors, while DVDs might have running times, actors, directors, etc. However, you also want to associate certain common types of data with both books and DVDs. For example, all library items might have Dewey decimal classifications as well as unique identification numbers assigned by the library. And every sort of library item has a title or name. In such a case, it can be advantageous to define a class that generalizes the commonality of all library items:

```
package org.example.library.collection {
    public class LibraryItem {
        protected var _ddc:String;
        protected var _id:String;
        protected var _name:String;
    }
}
```

```

public function LibraryItem( ) {}

public function setDdc(value:String):void {
    _ddc = value;
}
public function getDdc( ):String {
    return _ddc;
}

public function setId(value:String):void {
    _id = value;
}
public function getId( ):String {
    return _id;
}

public function setName(value:String):void {
    _name = value;
}
public function getName( ):String {
    return _name;
}
}
}

```

Then you can say that books and DVDs are both types of *LibraryItem*. It would then be appropriate to define a *Book* class and a *DVD* class that are subclasses of *LibraryItem*. The *Book* class might look like the following:

```

package org.examplelibrary.collection {
    import org.examplelibrary.collection.LibraryItem;
    public class Book extends LibraryItem {
        private var _authors:Array;
        private var _pageCount:uint;

        public function Book( ) {}

        public function setAuthors(value:Array):void {
            _authors = value;
        }
        public function getAuthors( ):Array {
            return _authors;
        }

        public function setPageCount(value:uint):void {
            _pageCount = value;
        }
        public function getPageCount( ):uint {
            return _pageCount;
        }
    }
}

```

The “Is a” and “Has a” test is helpful, but not always definitive in determining the relationship between classes. Often composition can be used even when inheritance would be acceptable and appropriate. In such cases the developer might opt for composition because it offers an advantage or flexibility not provided by inheritance. Furthermore, there are times when a class may appear to pass the “Is a” test yet inheritance would not be the correct relationship. For example, the library application might allow users to have accounts, and to represent the user, you would define a *User* class. The application might differentiate between types of users; for example, administrator and standard users. You could define *Administrator* and *StandardUser* classes. In such a case, the classes would appear to pass the “Is a” test in relation to *User*. It would seem to make sense that an *Administrator* is a *User*. However, if you consider the context an *Administrator* isn’t actually a *User*, but more appropriately an *Administrator* is a role for a *User*. If possible, it would be better to define *User* so it has a role of type *Administrator* or *StandardUser*.

By default it’s possible to extend any class. However you may want to ensure that certain classes are never subclassed. For this reason you can add the `final` attribute to the class declaration, as follows:

```
final public class Example
```

## Implementing Subclass Versions of Superclass Methods

### Problem

You want to implement a method in a subclass differently than how it was implemented in the superclass.

### Solution

The superclass method must be declared as `public` or `protected`. Use the `override` attribute when declaring the subclass implementation.

### Discussion

Often a subclass inherits all superclass methods directly without making any changes to the implementations. In those cases, the method is not redeclared in the subclass. However, there are cases in which a subclass implements a method differently than the superclass. When that occurs, you must override the method. To do that, the method must be declared as `public` or `protected` in the superclass. You can then declare the method in the subclass using the `override` attribute. As an example, you’ll first define a class, *Superclass*:

```
package {
    public class Superclass {
        public function Superclass() {}
        public function toString():String {
            return "Superclass.toString()";
        }
    }
}
```

Next, define *Subclass* so it inherits from *Superclass*:

```

package {
    public class Subclass extends Superclass {
        public function Subclass( ) {}
    }
}

```

By default, *Subclass* inherits the *toString()* method as it's implemented in *Superclass*:

```

var example:Subclass = new Subclass( );
trace(example.toString()); // Displays: Superclass.toString( )

```

If you want the *toString()* method of *Subclass* to return a different value, you'll need to override it in the subclass, as follows:

```

package {
    public class Subclass extends Superclass {
        public function Subclass( ) {}
        override public function toString():String {
            return "Subclass.toString( )";
        }
    }
}

```

When overriding a method, it must have exactly the same signature as the superclass. That means the number and type of parameters and the return type of the subclass override must be exactly the same as the superclass. If they aren't identical, the compiler throws an error.

Sometimes when you override a method you want the subclass implementation to be entirely different from the superclass implementation. However, sometimes you simply want to add to the superclass implementation. In such cases, you can call the superclass implementation from the subclass implementation using the `super` keyword to reference the superclass:

```

super.methodName( );

```

See Also

*Recipe 2.5*

## Creating Constants

### Problem

You want to declare a constant.

### Solution

Declare it just like you would declare a property, except use the `const` keyword in place of `var`.

### Discussion

As the name *constant* implies, constant values do not change. Constants are useful when you have complex values that you want to be able to reference by a simple identifier or when you want to be able to use compile-time error checking for values. `Math.PI` is an example of a constant that contains a complex value (which is the value of pi, or 3.14159). `MouseEvent.MOUSE_UP`, which

contains the value `mouseUp`, is an example of a constant that allows you to use error-checking. When you add an event listener for the mouse up event, you can use the string value `mouseUp`. However, if you accidentally have a typo, you won't be notified of an error, and your code won't work as expected:

```
// This is valid code, but because of the typo (mousUp instead of mouseUp) the
// code won't work as expected.
addEventListener("mousUp", onMouseUp);
```

Using a constant helps. If you accidentally misspell the constant, you will receive a compile error that helps you track down the error:

```
// This causes a compile error.
addEventListener(MouseEvent.CLICK, onMouseUp);
```

The syntax for declaring a constant is very similar to that for declaring a standard property. However, rather than using the `var` keyword you use the `const` keyword. Although not required, the majority of constants also happen to be public and static. If you want a constant to be public and static, you must use the correct attributes. Additionally, you must assign a value for a constant when declaring it:

```
static public const EXAMPLE:String = "example";
```

By convention, constant names are all in uppercase. This convention makes it easy to identify and differentiate constants from properties.

See Also

*Recipe 2.4*

## Dispatching Events

Problem

You want to dispatch events.

Solution

Extend *flash.events.EventDispatcher* and call the *dispatchEvent()* method.

Discussion

Events are an important way for objects to communicate. They are essential for creating flexible systems. Flash Player 9, for example, has a built-in event dispatching mechanism in the *flash.events.EventDispatcher* class. All classes that dispatch events inherit from *EventDispatcher* (e.g., *NetStream* and *Sprite*). If you want to define a class that dispatches events, you can extend *EventDispatcher*, as follows:

```
package {
    import flash.events.EventDispatcher;
    public class Example extends EventDispatcher {
```

```
}  
}
```

The *EventDispatcher* class has public methods called *addEventListener()* and *removeEventListener()* that you can call from any instance of an *EventDispatcher* subclass to register event listeners. *EventDispatcher* also defines a protected method called *dispatchEvent()*, which you can call from within a subclass to dispatch an event. The *dispatchEvent()* method requires at least one parameter as a *flash.events.Event* object or a subclass of *Event*.

# Index

-  
-source-path option (mxmml), 8

.  
(dot)  
  saving class files, 4  
.as files  
  custom classes, creating, 3  
.fla files, 3  
  class files, saving, 7

A  
addEventListener() method, 16

C  
classes  
  custom, 3  
  creating, 3  
  naming, 5  
  saving, 7  
command-line compilers  
  classpath, setting, 8  
compilers  
  saving class files, 7  
composition of classes, 11  
const keyword, 14  
constants, 14  
custom classes, 3  
  creating, 3

D  
declaring methods, 6  
dependencies of classes, 4  
directories  
  saving class files, 7  
dispatchEvent(), 16  
dots (.)  
  saving class files, 4

E  
EventDispatcher class, 16  
events  
  dispatching, 15  
extends keyword, 11

F  
flash.events, 16  
Flex Builder, 3  
  class files, saving, 7  
fromCharCode() method, 10  
function keyword, 6

I  
inheritance, defining, 11  
internal keyword, 6

M  
Macromedia XML (MXML), 3  
  class files, saving, 7  
Math class, 10  
Math.PI, 14  
MouseEvent.MOUSE\_UP, 14  
MXML (Macromedia XML), 3  
  class files, saving, 7  
mxmml, 7

N  
names of packages, 4

O  
override attribute, 13

P  
packages, 3  
private keyword, 5  
protected keyword, 6  
  subclasses, creating, 11  
  superclass methods and, 13  
public keyword, 5  
  subclasses, creating, 11  
  superclass methods and, 13

R  
round() method, 10

S  
static attribute, 10  
static methods  
  creating, 10  
static properties, 10  
subclasses  
  creating, 11  
  superclass methods, implementing, 13  
subdirectories, saving class files, 4  
super keyword, 14  
superclasses, 11  
  implementing subclass versions of, 13

T  
toString() method, 14

U  
underscore (  ), declaring private/protected property names, 6

V  
var keyword  
constants, creating, 14

         (underscore), declaring private/protected property names,  
6