

ActionScript 3.0 Cookbook

ActionScript Basics

By Joey Lott, Darron Schall and Keith Peters

Publisher: O'Reilly Media, Inc.

Pub Date: 2006-10-11

ISBN: 9780596526955

Table Of Contents

ActionScript Basics.....	3
Introduction	3
Creating an ActionScript Project	3
Customizing the Properties of an Application	4
Where to Place ActionScript Code	6
How to Trace a Message	9
Handling Events	10
Responding to Mouse and Key Events	12
Using Mathematical Operators	14
Checking Equality or Comparing Values	16
Performing Actions Conditionally	19
Performing Complex Conditional Testing	23
Repeating an Operation Many Times	25
Repeating a Task over Time	28
Creating Reusable Code	31
Generalizing a Method to Enhance Reusability	32
Exiting a Method	34
Obtaining the Result of a Method	35
Handling Errors	36
Index.....	41

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.

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.

ActionScript Basics

Introduction

Using ActionScript, you can create Flash applications that do just about anything you can imagine. But before launching into the vast possibilities, let's start with the basic foundation. The good news is that ActionScript commands follow a well-defined pattern, sharing similar syntax, structure, and concepts. Mastering the fundamental grammar puts you well on the way to mastering ActionScript.

This chapter addresses the frequent tasks and problems that relate to core ActionScript knowledge. Whether you are a beginner or master—or somewhere in between—these recipes help you handle situations that arise in every ActionScript project.

This book assumes that you have obtained a copy of Flex Builder 2 and have successfully installed it on your computer. It's also helpful if you have some experience using a previous version of ActionScript as well.

When you launch Flex Builder 2, the Eclipse IDE should start up and present you with a welcome screen. You are presented with various options to get started and more information about Flex and ActionScript 3, such as links to documentation, tutorials, and more. You can close that screen by clicking on the small “x” on its tab. Now you are in the Eclipse IDE itself, ready to start coding; but where do you go from here?

Flex Builder 2 allows you to create three kinds of projects: a Flex project, Flex Library project, and an ActionScript project. The difference is that Flex projects have access to the entire Flex Framework, which includes all of the Flex components, layout management, transitions, styles, themes, data binding, and all the other stuff that goes into making a Flex Rich Internet Application. Flex applications are written in MXML (a form of XML), which describes the layout and relationship between components. They use ActionScript for their business logic. Although you can use the ActionScript knowledge you learn from here in Flex applications you write, this book concentrates on ActionScript projects exclusively.

Now, if you are familiar with Flash 8 or earlier versions of the Flash IDE, you may be a bit baffled the first time you open up Flex Builder 2. There is no timeline, no library, no drawing tools or color pickers. You'll be doing pretty much everything by code alone, which is why it is called an *ActionScript project*, rather than a *Flash project*. So we'll first cover how to create a project and then to get you started with entering your first ActionScript statements.

Creating an ActionScript Project

Problem

You've launched Flex Builder 2 and want to create an ActionScript project.

Solution

Use the New ActionScript Project Wizard to set up your project.

Discussion

An ActionScript project usually consists of at least one class file and a folder named *bin* that contains the SWF and HTML files output by the compiler. It also consists of a lot of internal settings to let the compiler know where everything is and how to compile it all. Flex Builder 2 takes care of most of this for you when you use the New ActionScript Project Wizard. There are a few ways to start this wizard. You can use the menu File > New > ActionScript Project, or you can click on the New button in the top-right corner and select ActionScript Project from the list of available projects there. You can also click the small arrow next to the New button, which gives you the same list.

Whichever route you take to get there, you should wind up with the New ActionScript Project Wizard. Here you'll be prompted to type in a name for your project, such as *ExampleApplication*. Once you've created the project, you'll notice that the main application file is automatically set to the same name as the project name, with a *.as* extension.

Clicking the Next button gives you the opportunity to set custom class paths, additional libraries, and specify your output folder to something than the default *bin*. For now, you don't need to do anything here, so just press Finish to exit the wizard.

Flex Builder 2 now creates the necessary folders and files and applies all the default compiler settings for your project. In the Navigator view, you should now see a *ExampleApplication* project, which contains an empty *bin* folder and a *ExampleApplication.as* class file. Note that it has created this main class file for you automatically and has opened it up for editing in the code view. Also, in the Outline view, you can see a tree representation of the class, including its methods, properties, and any import statements.

To run your new application, you can press one of two buttons in the toolbar. One has a bug-like icon, which when pressed debugs the application, meaning it includes some extra information for debugging purposes and allows the use of trace statements. The button next to it—a circle with an arrow—runs the application. Both actions will create a *.swf* file and an HTML file, and then launch the HTML file in your default browser.

Of course, at this point, you haven't added anything to the application, so it is the equivalent of testing a blank *.fla* file in the Flash IDE. But go ahead and do so just to verify that everything is set up properly. You should get an empty web page with a blue background.

Customizing the Properties of an Application

Problem

You want to change the dimensions of the output *.swf*, or its background color, frame rate, etc.

Solution

Specify properties as ActionScript Compiler arguments or metadata in the class file.

Discussion

Unlike earlier versions of Flash, the ActionScript 3.0 compiler is actually a command-line compiler. Technically, you could create all your classes and directories and run the compiler from the command line with a long chain of parameters. However, it's much easier to let Eclipse keep track of all those parameters, add all of them, and run the compiler when you tell it to run.

When you create a new ActionScript project, it sets up default parameters that result in an 500×375 pixel *.swf*, with a frame rate of 24 frames per second (fps) and that blue background color you've seen. You can change any of these settings and many more. As you might expect, there are a few different ways to do this.

The first way to change compiler settings is to set the ActionScript compiler arguments. You do this by right-clicking on the project in the Navigator view and choosing Properties from the menu. Next, choose ActionScript Compiler from the list on the left. This allows you to change several aspects of how the compiler does its job. Look for the text field labeled "Additional compiler arguments." Anything you type in this text field is passed directly to the command-line compiler as an argument.

Here are the most common arguments you will probably be using:

`-default-t-size width height`

`-default-t-background-color color`

`-default-t-frame-rate fps`

You enter them exactly as presented, with numbers for arguments, like so:

`-default-t-size 800 600`

`-default-t-background-color 0xffffffff`

`-default-t-frame-rate 31`

The first example sets the resulting size of the resulting *.swf* to 800×600 pixels. The second sets its background to white, and the last sets its frame rate to 31 fps. Multiple arguments would just be placed one after the other on the same line, like so:

`-default-t-size 800 600 -default-t-frame-rate 31`



Check the Flex Builder 2 help files for *mxmlc options* to see the full list of command-line arguments you can enter here.

The second way to change these properties is through metadata in your main class file. Metadata consists of any statements that are not directly interpreted as ActionScript, but which the compiler uses to determine how to compile the final output files. The metadata statement that is equivalent to the previous example looks like this:

```
[SWF(width="800", height="600", backgroundColor="#ffffff", frameRate="31")]
```

This line is placed inside the main package block, but outside any class definitions (usually just before or after any import statements).

Where to Place ActionScript Code

Problem

You have a new ActionScript project and need to know where to put the code for it to execute properly.

Solution

Place ActionScript code in the constructor and additional methods of the class.

Discussion

In ActionScript 1.0 and 2.0, you had many choices as to where to place your code: on the timeline, on buttons and movie clips, on the timeline of movie clips, in external `.as` files referenced with *#include*, or as external class files. ActionScript 3.0 is completely class-based, so all code must be placed in methods of your project's classes.

When you create a new ActionScript project, the main class is automatically created, and opened in the Code view. It should look something like this:

```
package {
    import flash.display.Sprite;

    public class ExampleApplication extends Sprite
    {
        public function ExampleApplication()
        {

        }
    }
}
```

Even if you are familiar with classes in ActionScript 2.0, there are some new things here. There is a lot more information on this subject in *Chapter 2*, but let's go through the basics here.

The first thing you'll notice is the word *package* at the top of the code listing. Packages are used to group classes of associated functionality together. In ActionScript 2.0, packages were inferred through the directory structure used to hold the class files. In ActionScript 3.0, however, you must explicitly specify packages. For example, you could have a package of utility classes. This would be declared like so:

```
package com.as3cb.utils {

}
```

If you don't specify a package name, your class is created in the default, top-level package. You should still include the *package* keyword and braces.

Next, place any `import` statements. Importing a class makes that class available to the code in the file and sets up a shortcut so you don't have to type the full package name every time you want to refer to that class. For example, you can use the following `import` statement:

```
import com.as3cb.utils.StringUtils;
```

Thereafter you can refer to the *StringUtil* class directly without typing the rest of the path. As shown in the earlier example, you will need to import the *Sprite* class from the *flash.display* package, as the default class extends the *Sprite* class.

Next up is the main class, *ExampleApplication*. You might notice the keyword *public* in front of the class definition. Although you can't have private classes within a package, you should label the class public. Note that the main class extends *Sprite*. Also, a *.swf* itself is a type of sprite or movie clip, which is why you can load a *.swf* into another *.swf* and largely treat it as if it were just another nested sprite or movie clip. This main class represents the *.swf* as a whole, so it should extend the *Sprite* class or any class that extends the *Sprite* class (such as *MovieClip*).

Finally, there is a public function (or method, in class terminology) with the same name as the class itself. This makes it a *constructor*. A class's constructor is automatically run as soon as an instance of the class is created. In this case, it is executed as soon as the *.swf* is loaded into the Flash player. So where do you put your code to get it to execute? Generally, you start out by putting some code in the constructor method. Here's a very simple example that just draws a bunch of random lines to the screen:

```
package {
    import flash.display.Sprite;
    public class ExampleApplication extends Sprite {
        public function ExampleApplication() {
            graphics.lineStyle(1, 0, 1);
            for(var i:int=0;i<100;i++) {
                graphics.lineTo(Math.random() * 400, Math.random() * 400);
            }
        }
    }
}
```

Save and run the application. Your browser should open the resulting HTML file and display the *.swf* with 100 random lines in it. As you can see, the constructor was executed as soon as the file was loaded into the player.

In practice, you usually want to keep code in the constructor to a bare minimum. Ideally the constructor would just contain a call to another method that initializes the application. See Recipes 1.13 and 1.14 for more on methods.

For beginners, now that you know where to enter code, here is quick primer on terminology. These definitions are briefly stated and intended to orient people who have never programmed before. For more complete definitions, refer to the Flash help files.

Variables

Variables are convenient placeholders for data in your code, and you can name them anything you'd like, provided the name isn't already reserved by ActionScript and the name starts with a letter, underscore, or dollar sign (but not a number). The help files installed with Flex Builder 2 contain a list of reserved words. Variables are convenient for holding interim information, such as a sum of numbers, or to refer to something, such as a text field or sprite. Variables are *declared* with the *var* keyword the first time they are used in a script. You can assign a value to a variable using an equal sign (=), which is also known as the *assignment operator*. If a variable is declared outside a class method, it is a *class variable*. Class variables, or *properties*, can have access modifiers, *public*, *private*, *protected*, or *internal*. A private variable can only be accessed from within the class itself, whereas public variables can be

accessed by objects of another class. Protected variables can be accessed from an instance of the class or an instance of any subclass, and internal variables can be accessed by any class within the same package. If no access modifier is specified, it defaults to internal.

Functions

Functions are blocks of code that do something. You can *call* or *invoke* a function (that is, execute it) by using its name. When a function is part of a class, it is referred to as a *method* of the class. Methods can use all the same modifiers as properties.

Scope

A variable's *scope* describes when and where the variable can be manipulated by the code in a movie. Scope defines a variable's life span and its accessibility to other blocks of code in a script. Scope determines how long a variable exists and from where in the code you can set or retrieve the variable's value. A function's scope determines where and when the function is accessible to other blocks of code. *Recipe 1.13* deals with issues of scope.

Event handler

A *handler* is a function or method that is executed in response to some event such as a mouseclick, a keystroke, or the movement of the playhead in the timeline.

Objects and classes

An *object* is something you can manipulate programmatically in ActionScript, such as a sprite. There are other types of objects, such as those used to manipulate colors, dates, and text fields. Objects are instances of *classes*, which means that a class is a template for creating objects and an object is a particular instance of that class. If you get confused, think of it in biological terms: you can consider yourself an object (instance) that belongs to the general class known as humans.

Methods

A *method* is a function associated with an object that operates on the object. For example, a text field object's *replaceSelectedText()* method can be used to replace the selected text in the field.

Properties

A *property* is an attribute of an object, which can be read and/or set. For example, a sprite's horizontal location is specified by its *x* property, which can be both tested and set. On the other hand, a text field's *length* property, which indicates the number of characters in the field, can be tested but cannot be set directly (it can be affected indirectly, however, by adding or removing text from the field).

Statements

ActionScript commands are entered as a series of one or more *statements*. A statement might tell the playhead to jump to a particular frame, or it might change the size of a sprite. Most ActionScript statements are terminated with a semicolon (;). This book uses the terms *statement* and *action* interchangeably.

Comments

Comments are notes within code that are intended for other humans and ignored by Flash. In ActionScript, single-line comments begin with // and terminate automatically at the end of the current line. Multiline comments begin with /* and are terminated with */.

Interpreter

The *ActionScript interpreter* is that portion of the Flash Player that examines your code and attempts to understand and execute it. Following ActionScript's strict rules of grammar ensures that the interpreter can easily understand your code. If the interpreter encounters an error, it often fails silently, simply refusing to execute the code rather than generating a specific error message.

Don't worry if you don't understand all the specifics. You can use each recipe's solution without understanding the technical details, and this primer should help you understand the terminology.

See Also

Recipes *1.13* and *1.14*

How to Trace a Message

Problem

You need to trace out a message or the value of some data at runtime.

Solution

Use the *trace* function, pass the data to it, run your application, and look for a message in the Console in Eclipse.

Discussion

You can trace out a message, the value of a variable, or just about any other data using *trace*, just as you would in earlier versions of ActionScript. Some examples:

```
trace("Hello, world");
```

```
trace(userName);
```

```
trace("My name is " + userName + ".");
```

Since the *.swf* is now launched in an external browser, it might seem that there is no way to capture the output of these trace statements. Fortunately, it is possible, and this functionality has been built in to Flex Builder 2 via the Console view. The Console view is the equivalent of the Output panel in the Flash IDE. Although it is not open when you first start Eclipse, it appears when needed.

The only requirement to using *trace* and the Console view is that you use Debug to test your application. Doing so includes extra features in the *.swf* that allows it to communicate back to the Console behind the scenes and pass any messages you trace. The following class creates a variable, assigns a value to it, and then traces it, along with some other string data:

```
package {
    import flash.display.Sprite;

    public class ExampleApplication extends Sprite {
        public function ExampleApplication() {
            var userName:String = "Bill Smith";
            trace("My name is " + userName + ".");
        }
    }
}
```

```
}  
}  
}
```

Now when you debug your application, it launches as usual in your default browser. Close the browser and switch back to Eclipse. You will see that the Console view is now open and has displayed the data you traced out.

When you launch the debug version of an application, you must have the debug version of Flash Player installed. If you don't have the debug version of Flash Player, you'll see an error message notifying you, and you'll have to download and install it from <http://www.adobe.com/support/flashplayer/downloads.html>.

Additionally, the debug version of Flash Player can write trace content to a file. The file that Flash Player uses is determined by *mm.cfg*, a file that is stored in the following locations:

Operating system	Location
Windows XP	C:\Documents and Settings\[<i>user name</i>]\mm.cfg
Windows 2000	C:\mm.cfg
Mac OS X	MachD:Library:Application Support:macromedia:mm.cfg

The *mm.cfg* file allows you to set the following variables:

TraceOutputFileEnable

The value can be 0 (don't write trace content to a file) or 1 (write to a file).

TraceOutputFileName

The path to the file to which to write. If a value isn't specified, then the content is written to *flashlog.txt* in the same directory as *mm.cfg*.

ErrorReportingEnable

The value can be 0 (don't write errors to the logfile) or 1 (write errors to the logfile). The default value is 0.

MaxWarnings

The maximum number of errors to write to the logfile. If this value is set to 0, there is no limit. If a larger value is specified, that limit is imposed and any errors beyond the limit are not written to the log.

At a minimum *mm.cfg* must contain the following enable writing to a file.

```
TraceOutputFileEnable=1
```

If you want to specify more than one variable, you should place each on a new line, as follows

```
TraceOutputFileEnable=1  
TraceOutputFileName=C:\flex.log
```

Handling Events

Problem

You want to have some code repeatedly execute.

Solution

Add a listener to the *enterFrame* event and assign a method as a handler.

Discussion

In ActionScript 2.0 handling the *enterFrame* event was quite simple. You just had to create a timeline function called *onEnterFrame* and it was automatically called each time a new frame began. In ActionScript 3.0, you have much more control over the various events in a *.swf*, but a little more work is required to access them.

If you are familiar with the *EventDispatcher* class from ActionScript 2.0, you should be right at home with ActionScript 3.0's method of handling events. In fact, *EventDispatcher* has graduated from being an externally defined class to being the base class for all interactive objects, such as sprites.

To respond to the *enterFrame* event, you have to tell your application to listen for that event and specify which method you want to be called when the event occurs. This is done with the *addEventListener* method, which is defined as follows:

```
addEventListener(type:String, listener:Function)
```



There are additional parameters you can look up in the help files, but this is the minimum implementation.

The *type* parameter is the type of event you want to listen to. In this case, it would be the string, "enterFrame". However, using string literals like that opens your code to errors that the compiler cannot catch. If you accidentally typed "enterFrane", for example, your application would simply listen for an "enterFrane" event. To guard against this, it is recommended that you use the static properties of the *Event* class. You should already have the *Event* class imported, so you can call the *addEventListener* method as follows:

```
addEventListener(Event.ENTER_FRAME, onEnterFrame);
```

Now if you accidentally typed *Event.ENTER_FRANE*, the compiler would complain that such a property did not exist.

The second parameter, *onEnterFrame*, refers to another method in the class. Note, that in ActionScript 3.0, there is no requirement that this method be named *onEnterFrame*. However, naming event handling methods on plus the event name is a common convention. This method gets passed an instance of the *Event* class when it is called. Therefore, you'll need to import that class and define the method so it accepts an event object:

```
import flash.events.Event;
```

```
private function onEnterFrame(event:Event) {  
  
}
```

The event object contains information regarding the event that may be useful in handling it. Even if you don't use it, you should still set your handler up to accept it. If you are familiar with the ActionScript 2.0 version of *EventDispatcher*, you'll see a difference in implementation here. In the earlier version, there was an issue with the scope of the function used to handle the event, which

often required the use of the *Delegate* class to correct. In ActionScript 3.0, the scope of the handling method remains the class of which it is a method, so there is no necessity to use *Delegate* to correct scope issues.

Here is a simple application that draws successive random lines, using all the concepts discussed in this recipe:

```
package {
    import flash.display.Sprite;
    import flash.events.Event;

    public class ExampleApplication extends Sprite {

        public function ExampleApplication() {
            graphics.lineStyle(1, 0, 1);
            addEventListener(Event.ENTER_FRAME, onEnterFrame);
        }

        private function onEnterFrame(event:Event):void {
            graphics.lineTo(Math.random() * 400, Math.random() * 400);
        }
    }
}
```

Responding to Mouse and Key Events

Problem

You want to do something in response to a mouse or keyboard action.

Solution

Listen for and handle mouse or key events.

Discussion

Handling mouse and key events is very similar to handling the *enterFrame* event, as discussed in the *Recipe 1.5*, but does require a little work. For mouse events, the main application class will not receive these directly, so it must listen for them on another object in the display list. (For a complete discussion of the display list, see *Chapter 5*.) The following example creates a sprite, adds it to the display list, and draws a rectangle in it:

```
package {
    import flash.display.Sprite;
    import flash.events.MouseEvent;

    public class ExampleApplication extends Sprite {
        private var _sprite:Sprite;

        public function ExampleApplication() {
            _sprite = new Sprite();
            addChild(_sprite);
            _sprite.graphics.beginFill(0xffffffff);
        }
    }
}
```

```

_sprite.graphics.drawRect(0, 0, 400, 400);
_sprite.graphics.endFill();

```

Note that the mouse event names are defined in the *MouseEvent* class, and the handler methods get passed an instance of the *MouseEvent* class, so you'll need to import that class. Then you can add mouse listeners to this sprite:

```

_sprite.addEventListener(MouseEvent.CLICK, onMouseDown);
_sprite.addEventListener(MouseEvent.CLICK, onMouseUp);
}

```

Next, define the two handler methods, `onMouseDown` and `onMouseUp`:

```

private function onMouseDown(event:MouseEvent):void {
    _sprite.graphics.lineStyle(1, 0, 1);
    _sprite.graphics.moveTo(mouseX, mouseY);
    _sprite.addEventListener(MouseEvent.CLICK, onMouseMove);
}

private function onMouseUp(event:MouseEvent):void
{
    _sprite.removeEventListener(MouseEvent.CLICK, onMouseMove);
}

```

The `onMouseDown` methods sets a drawing line style on the new sprite and moves the drawing cursor to the mouse position. It then adds yet a third mouse listener for the *MouseMove* event.

The `onMouseUp` methods removes that listener via the *removeEventListener* method. This has the same syntax as *addEventListener*, but tells the class to stop listening to the specified event.

Finally, define `onMouseMove` and close up the class and package:

```

private function onMouseMove(event:MouseEvent):void {
    _sprite.graphics.lineTo(mouseX, mouseY);
}
}
}

```

This creates a simple event-driven drawing program.

Keyboard events are a little easier to handle. The only requirement for listening and responding to keyboard events is that the object that receives the events must have focus. You do this for the main application by adding the line:

```
stage.focus = this;
```

The following example shows a simple class that listens for the *keyDown* event and traces out the character code for that key. This also demonstrates how to use some of the data contained in the event object passed to the handler method. Note that keyboard events use the class *KeyboardEvent*.

```

package {
    import flash.display.Sprite;
    import flash.events.KeyboardEvent;

    public class ExampleApplication extends Sprite {
        public function ExampleApplication() {
            stage.focus = this;
            addEventListener(KeyboardEvent.KEY_DOWN, onKeyDown);
        }
    }
}

```

```

    }
    private function onKeyDown(event:KeyboardEvent):void {
        trace("key down: " + event.charCode);
    }
}

```

See Also

Recipe 1.5

Using Mathematical Operators

Problem

You want to modify something over time, such as the rotation or position of a sprite.

Solution

Use the compound assignment operators to change a variable or property in increments; or, if incrementing or decrementing by one, use the prefix or postfix increment or decrement operators.

Discussion

Often you'll want the new value of a variable or property to depend on the previous value. For example, you might want to move a sprite to a new position that is 10 pixels to the right of its current position.

In an assignment statement—any statement using the assignment operator (an equals sign)—the expression to the right of the equals sign is evaluated and the result is stored in the variable or property on the left side. Therefore, you can modify the value of a variable in an expression on the right side of the equation and assign that new value to the very same variable on the left side of the equation.

Although the following may look strange to those who remember basic algebra, it is very common for a variable to be set equal to itself plus some number:

```

// Add 6 to the current value of quantity, and assign that new
// value back to quantity. For example, if quantity was 4, this
// statement sets it to 10.
quantity = quantity + 6;

```

However, when performing mathematical operations, it is often more convenient to use one of the *compound assignment operators*, which combine a mathematical operator with the assignment operator. The `+=`, `-=`, `*=`, and `/=` operators are the most prevalent compound assignment operators. When you use one of these compound assignment operators, the value on the right side of the assignment operator is added to, subtracted from, multiplied by, or divided into the value of the variable on the left, and the new value is assigned to the same variable. The following are a few examples of equivalent statements.

These statements both add 6 to the existing value of `quantity`:

```
quantity = quantity + 6;
quantity += 6;
```

These statements both subtract 6 from the existing value of quantity:

```
quantity = quantity - 6;
quantity -= 6;
```

These statements both multiple quantity by factor:

```
quantity = quantity * factor;
quantity *= factor;
```

These statements both divide quantity by factor:

```
quantity = quantity / factor;
quantity /= factor;
```

There should be no space between the two symbols that make up a compound assignment operator. Additionally, if you are incrementing or decrementing a variable by 1, you can use the increment or decrement operators.

This statement adds 1 to quantity:

```
quantity++;
```

and has the same effect as either of these statements:

```
quantity = quantity + 1;
quantity += 1;
```

This statement subtracts 1 from quantity:

```
quantity --;
```

and has the same effect as either of these statements:

```
quantity = quantity - 1;
quantity -= 1;
```

You can use the increment and decrement operators (-- and ++) either before or after the variable or property they operate upon. If used before the operand, they are called *prefix operators*. If used after the operand, they are called *postfix operators*. The prefix and postfix operators modify the operand in the same way but at different times. In some circumstances, there is no net difference in their operation, but the distinction is still important in many cases. When using prefix operators, the value is modified before the remainder of the statement or expression is evaluated. And if you're using postfix operators, the value is modified after the remainder of the statement has executed. Note how the first example increments quantity after displaying its value, whereas the second example increments quantity before displaying its value:

```
var quantity:Number = 5;
trace(quantity++); // Displays: 5
trace(quantity);  // Displays: 6
```

```
var quantity:Number = 5;
trace(++quantity); // Displays: 6
trace(quantity);   // Displays: 6
```

Getting back to the original problem, you can use these operators to modify a property over time. This example causes the specified sprite to rotate by five degrees each time the method is called:

```
private function onEnterFrame(event:Event) {  
    _sprite.rotation += 5;  
}
```

Note that in ActionScript 3.0, you would have to add an event listener to the `enterFrame` event and set this method as the event handler for this to work properly. See *Recipe 1.5* for information on how to handle the `enterFrame` event.

See Also

Recipe 1.5

Checking Equality or Comparing Values

Problem

You want to check if two values are equal.

Solution

Use the equality (or inequality) or strict equality (or strict inequality) operator to compare two values. To check whether a value is a valid number, use `isNaN()`.

Discussion

Equality expressions always return a Boolean value indicating whether the two values are equal. The equality (and inequality) operators come in both regular and strict flavors. The regular equality and inequality operators check whether the two expressions being compared can be resolved to the same value after converting them to the same datatype. For example, note that the string “6” and the number 6 are considered equal because the string “6” is converted to the number 6 before comparison:

```
trace(5 == 6);    // Displays: false  
trace(6 == 6);    // Displays: true  
trace(6 == "6"); // Displays: true  
trace(5 == "6"); // Displays: false
```

Note that in a project with default settings, the previous code example won’t even compile. That’s because it is compiled with a `strict` flag, causing the compiler to be more exact in checking datatypes at compile time. It complains that it is being asked to compare an *int* with a *String*. To turn off the strict flag, go to the ActionScript Compiler section of the project’s properties, and uncheck the box next to “Enable compile-time type checking (-strict)”. It is suggested, however, that you leave this option on for most projects, as it gives you better protection against inadvertent errors.

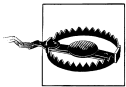
The logical inequality operator (`!=`) returns `false` if two values are equal and `true` if they aren’t. If necessary, the operands are converted to the same datatype before the comparison:

```
trace(5 != 6);    // Displays: true  
trace(6 != 6);    // Displays: false  
trace(6 != "6"); // Displays: false  
trace(5 != "6"); // Displays: true
```

Again, this example only compiles if strict type checking is disabled.

On the other hand, if you have turned off the strict flag, but you want to perform a strict comparison in one section of code, you can use the strict equality and inequality operators, `===` and `!==`. These first check whether the values being compared are of the same datatype before performing the comparison. Differences in datatypes causes the strict equality operator to return false and the strict inequality operator to return true:

```
trace(6 === 6);    // Displays: true
trace(6 === "6"); // Displays: false
trace(6 !== 6);   // Displays: false
trace(6 !== "6"); // Displays: true
```



There is a big difference between the assignment operator (`=`) and the equality operator (`==`). If you use the assignment operator instead of the equality operator, the variable's value will change rather than testing its current value.

Using the wrong operator leads to unexpected results. In the following example, `quantity` equals 5 at first, so you might expect the subsequent `if` statement to always evaluate to false, preventing the `trace()` from being executed:

```
var quantity:int = 5;
// The following code is wrong. It should be if (quantity == 6) instead
if (quantity = 6) {
    trace("Rabbits are bunnies.");
}
trace("quantity is " + quantity); // Displays: quantity is 6
```

However, the example mistakenly uses the assignment operator (`=`) instead of the equality operator (`==`). That is, the expression `quantity = 6` sets `quantity` to 6 instead of testing whether `quantity` is 6. When used in an `if` clause, the expression `quantity = 6` is treated as the number 6. Because, any nonzero number used in a test expression converts to the Boolean true, the `trace()` action is called. Replace the test expression with `quantity == 6` instead. Fortunately, the ActionScript 3.0 compiler is smart enough to recognize this common error and although the code still compiles, you aren't given a warning.

You can check an item's datatype using the `is` operator, as follows:

```
var quantity:int = 5;
if (quantity is int) {
    trace("Yippee. It's an integer.");
}
```



Note that the new ActionScript 3.0 types `int` and `uint` will also test positive as `Numbers`.

However, some numeric values are invalid. The following example results in `quantity` being set equal to `NaN` (a constant representing invalid numbers, short for not a number) because the calculation cannot be performed in a meaningful way:

```
var quantity:Number = 15 - "rabbits";
```

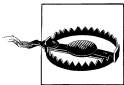
Despite its name, `NaN` is a recognized value of the `Number` datatype:

```
trace(typeof quantity); // Displays: "number"
```

Therefore, to test if something is not just any number, but a valid number, try this:

```
var quantity:Number = 15 - "rabbits";
if (quantity is Number) {

    // Nice try, but this won't work
    if (quantity != NaN) {
        trace("Yippee. It's a number.");
    }
}
```



However, you can't simply compare a value to the constant NaN to check whether it is a valid number. The ActionScript 3.0 compiler even gives you a warning to this effect. Instead, you must use the special *isNaN()* function to perform the test.

To determine if a number is invalid, use the special *isNaN()* function, as follows:

```
var quantity:Number = 15 - "rabbits";
if (isNaN(quantity)) {
    trace("Sorry, that is not a valid number.");
}
```

To test the opposite of a condition (i.e., whether a condition is not true) use the logical *NOT* operator (!). For example, to check whether a variable contains a *valid* number, use *!isNaN()*, as follows:

```
var quantity:Number = 15 - "rabbits";
if (!isNaN(quantity)) {

    // The number is not invalid, so it must be a valid number
    trace ("That is a valid number.");
}
```

Of course, you can perform comparisons using the well-known comparison operators. For example, you can use the `<` and `>` operators to check if one value is less than or greater than another value:

```
trace(5 < 6); // Displays: true
trace(5 > 5); // Displays: false
```

Similarly, you can use the `<=` and `>=` operators to check if one value is less than or equal to, or greater than or equal to, another value:

```
trace(5 <= 6); // Displays: true
trace(5 >= 5); // Displays: true
```

You should also be aware that ActionScript compares datatypes differently. ActionScript datatypes can be categorized either as *primitive* (*string*, *number*, and *Boolean*) or *composite* (*object*, *sprite*, and *array*). When you compare primitive datatypes, ActionScript compares them “by value.” In this example, *quantity* and *total* are considered equal because they both contain the value 6:

```
var quantity:Number = 6;
var total:Number = 6;
trace (quantity == total); // Displays: true
```

However, when you compare composite datatypes, ActionScript compares them “by reference.” Comparing items by reference means that the two items are considered equal only if both point to exactly the same object, not merely objects with matching contents. For example, two arrays containing exactly the same values are not considered equal:

```
// Create two arrays with the same elements.
var arrayOne:Array = new Array("a", "b", "c");
var arrayTwo:Array = new Array("a", "b", "c");
trace(arrayOne == arrayTwo);           // Displays: false
```

Two composite items are equal only if they both refer to the identical object, array, or sprite. For example:

```
// Create a single array
var arrayOne:Array = new Array("a", "b", "c");
// Create another variable that references the same array.
var arrayTwo:Array = arrayOne;
trace(arrayOne == arrayTwo);           // Displays: true
```

See Also

Recipe 5.8

Performing Actions Conditionally

Problem

You want to perform some action only when a condition is true.

Solution

Use an *if* or a *switch* statement.

Discussion

You often need your ActionScript code to make decisions, such as whether to execute a particular action or group of actions. To execute some action under certain circumstances, use one of ActionScript’s *conditional* statements: *if*, *switch*, or the ternary conditional operator (? :).

Conditional statements allow you to make logical decisions, and you’ll learn from experience which is more appropriate for a given situation. For example, the *if* statement is most appropriate when you want to tell a Flash movie to do something only when a certain condition is met (e.g., when the condition is true). When you have several possible conditions to test, you can use the *switch* statement instead. And you can use Flash’s ternary conditional operator to perform conditional checking and assignment on a single line.

First let’s look at the *if* statement. Of the conditional statements in ActionScript, the *if* statement is the most important to understand. In its most basic form, an *if* statement includes the keyword *if* followed by the test expression whose truthfulness you want to evaluate to determine which action or actions to execute. The test expression must be in parentheses and the statement(s) to be executed should be within curly braces (the latter is mandatory if there is more than one statement in the statement block).

Here we check whether `animalName` contains the word “turtle.” This might be used to check whether the user answered a quiz question correctly (here, `animalName` is a variable assumed to contain the user’s answer). Note that the double equals sign (`==`) is used to test whether two items are equal. It should not be confused with the single equals sign (`=`), which is used to assign a value to an item.

```
if (animalName == "turtle") {  
    // This trace() statement executes only when animalName is equal  
    // to "turtle".  
    trace("Yay! 'Turtle' is the correct answer.");  
}
```

Additionally, you can add an *else* clause to an *if* statement to perform alternative actions if the condition is false. Note that for the *trace()* command to have any effect, the *.swf* must be compiled using Debug, and not Run, mode. Make a call to a method named *showMessage()* that displays an appropriate message depending on whether the user got the answer right or wrong:

```
if (animalName == "turtle") {  
    // These statements execute only when animalName is equal  
    // to "turtle".  
    showMessage("Yay! 'Turtle' is the correct answer.");  
}  
else {  
    // These statements execute only when animalName is not equal  
    // to "turtle".  
    showMessage("Sorry, you got the question wrong.");  
}
```



For testing purposes, you can create a *showMessage()* method that traces out the string sent to it. In a real-world example, you might want to display this message in a text field, or display it to the user some other way, such as in a dialog box.

You can add an *else if* clause to an *if* statement. If the *if* condition is true, the *else if* clause is skipped. If the *if* condition is false, the ActionScript interpreter checks to see if the *else if* condition is true:

```
if (animalName == "turtle") {  
    // This trace() statement executes only when animalName is equal  
    // to "turtle".  
    showMessage ("Yay! 'Turtle' is the correct answer.");  
}  
else if (animalName == "dove") {  
    // This trace() statement executes only when animalName is not  
    // "turtle", but is "dove".  
    showMessage ("Sorry, a dove is a bird, not a reptile.");  
}
```

What if the preceding example was written as two separate *if* statements (one to check if `animalName` is “turtle” and another to check if it is “dove”)? The example would work as intended, but it would be less efficient. Using the *else if* statement guarantees that if `animalName` is “turtle”; we don’t bother checking if it is also equal to “dove.”



If your two conditions are mutually exclusive, use an *else if* clause to check the second condition. If your two conditions are not mutually exclusive, and you want to perform both statement blocks when both conditions are met, use two separate *if* statements.

When you use an *if* statement with both *else if* and *else* clauses, the *else* clause must be the last clause in the statement. The final *else* clause is convenient as a catchall; it's where you can put statements that take the appropriate action if none of the other conditions are met.

```
if (animalName == "turtle") {
    // This trace() statement executes only when animalName is equal
    // to "turtle".
    showMessage ("Yay! 'Turtle' is the correct answer.");
}
else if (animalName == "dove") {
    // This statement executes only when animalName is not
    // "turtle", but is "dove".
    showMessage ("Sorry, a dove is a bird, not a reptile.");
}
else {
    // This statement executes only when animalName is neither
    // "turtle" nor "dove".
    showMessage ("Sorry, try again.");
}
```

You can also include more than one *else if* clause in an *if* statement. However, in that case, you should most likely use a *switch* statement instead; generally, *switch* statements are more legible and succinct than the comparable *if* statement. Where performance is critical, some ActionScripters prefer to use *if* statements, which allow somewhat greater control for optimization purposes.

A *switch* statement is composed of three parts:

The switch keyword

Every *switch* statement must begin with the `switch` keyword.

Test expression

An expression, enclosed in parentheses, whose value you want to test to determine which action or actions to execute.

The switch statement body

The statement body, enclosed in curly braces, is composed of *cases*. Each case is made up of the following parts:

The case or default keyword

A case must begin with a `case` keyword. The exception is the default case (analogous to an *else* clause in an *if* statement), which uses the `default` keyword.

Case expression

An expression, whose value is to be compared to the *switch* statement's test expression. If the two values are equal, the code in the case body is executed. The default case (the case that uses the `default` keyword) does not need a case expression.

Case body

One or more statements, usually ending in a `break` statement, to be performed if that *case* is true.

The `switch` keyword is always followed by the test expression in parentheses. Then the *switch* statement body is enclosed in curly braces. There can be one or more *case* statements within the *switch* statement body. Each case (other than the default case) starts with the case keyword followed by the case expression and a colon. The default case (if one is included) starts with the `default` keyword followed by a colon. Therefore, the general form of a *switch* statement is:

```
switch (testExpression) {
    case caseExpression:
        // case body
    case caseExpression:
        // case body
    default:
        // case body
}
```



Note that once a case tests true, all the remaining actions in all subsequent cases within the *switch* statement body also execute. This example is most likely not what the programmer intended.

Here is an example:

```
var animalName:String = "dove";

/* In the following switch statement, the first trace() statement
   does not execute because animalName is not equal to "turtle".
   But both the second and third trace() statements execute,
   because once the "dove" case tests true, all subsequent code
   is executed.
*/
switch (animalName) {
    case "turtle":
        trace("Yay! 'Turtle' is the correct answer.");
    case "dove":
        trace("Sorry, a dove is a bird, not a reptile.");
    default:
        trace("Sorry, try again.");
}
```

Normally, you should use *break* statements at the end of each case body to exit the *switch* statement after executing the actions under the matching case.



The *break* statement terminates the current *switch* statement, preventing statements in subsequent case bodies from being erroneously executed.

You don't need to add a *break* statement to the end of the last *case* or *default* clause, since it is the end of the *switch* statement anyway.

```
var animalName:String = "dove";

// Now, only the second trace() statement executes.
switch (animalName) {
    case "turtle":
```

```

        trace("Yay! 'Turtle' is the correct answer.");
        break;
    case "dove":
        trace("Sorry, a dove is a bird, not a reptile.");
        break;
    default:
        trace("Sorry, try again.");
}

```

The *switch* statement is especially useful when you want to perform the same action for one of several matching possibilities. Simply list multiple case expressions one after the other. For example:

```

switch (animalName) {
    case "turtle":
    case "alligator":
    case "iguana":
        trace("Yay! You named a reptile.");
        break;
    case "dove":
    case "pigeon":
    case "cardinal":
        trace("Sorry, you specified a bird, not a reptile.");
        break;
    default:
        trace("Sorry, try again.");
}

```

ActionScript also supports the ternary conditional operator (`? :`), which allows you to perform a conditional test and an assignment statement on a single line. A *ternary operator* requires three operands, as opposed to the one or two operands required by unary and binary operators. The first operand of the conditional operator is a conditional expression that evaluates to either true or false. The second operand is the value to assign to the variable if the condition is true, and the third operand is the value to assign if the condition is false.

$$\text{varName} = (\text{conditional expression}) ? \text{valueIfTrue} : \text{valueIfFalse};$$

Performing Complex Conditional Testing

Problem

You want to make a decision based on multiple conditions.

Solution

Use the logical *AND* (`&&`), *OR* (`||`), and *NOT* (`!`) operators to create compound conditional statements.

Discussion

Many statements in ActionScript can involve conditional expressions, including *if*, *while*, and *for* statements, and statements using the ternary conditional operator. To test whether two

conditions are both true, use the logical *AND* operator, `&&`, as follows (see *Chapter 14* for details on working with dates):

```
// Check if today is April 17th.
var current:Date = new Date();
if (current.getDate() == 17 && current.getMonth() == 3) {
    trace ("Happy Birthday, Bruce!");
}
```

You can add extra parentheses to make the logic more apparent:

```
// Check if today is April 17th.
if ((current.getDate() == 17) && (current.getMonth() == 3)) {
    trace ("Happy Birthday, Bruce!");
}
```

Here we use the logical *OR* operator, `||`, to test whether either condition is true:

```
// Check if it is a weekend.
if ((current.getDay() == 0) || (current.getDay() == 6)) {
    trace ("Why are you working on a weekend?");
}
```

You can also use a logical *NOT* operator, `!`, to check if a condition is not true:

```
// Check to see if the name is not Bruce.
if (!(userName == "Bruce")) {
    trace ("This application knows only Bruce's birthday.");
}
```

The preceding example could be rewritten using the inequality operator, `!=`:

```
if (userName != "Bruce") {
    trace ("This application knows only Bruce's birthday.");
}
```

Any Boolean value, or an expression that converts to a Boolean, can be used as the test condition:

```
// Check to see if a sprite is visible. If so, display a
// message. This condition is shorthand for _sprite.visible == true
if (_sprite.visible) {
    trace("The sprite is visible.");
}
```

The logical *NOT* operator is often used to check if something is false instead of true:

```
// Check to see if a sprite is invisible (not visible). If so,
// display a message. This condition is shorthand for
// _sprite.visible != true or _sprite.visible == false.
if (!_sprite.visible) {
    trace("The sprite is invisible. Set it to visible before trying this action.");
}
```

The logical *NOT* operator is often used in compound conditions along with the logical *OR* operator:

```
// Check to see if the name is neither Bruce nor Joey. (This could
// also be rewritten using two inequality operators and a logical
// AND.)
if (!(userName == "Bruce" || userName == "Joey")) {
```

```
    trace ("Sorry, but only Bruce and Joey have access to this application.");  
}
```

ActionScript doesn't bother to evaluate the second half of a logical *AND* statement unless the first half of the expression is true. If the first half is false, the overall expression is always false, so it would be inefficient to bother evaluating the second half. Likewise, ActionScript does not bother to evaluate the second half of a logical *OR* statement unless the first half of the expression is false. If the first half is true, the overall expression is always true.

Repeating an Operation Many Times

Problem

You want to perform some task multiple times within a single frame.

Solution

Use a looping statement to perform the same task multiple times within a single frame. For example, you can use a *for* statement:

```
for (var i:int = 0; i < 10; i++) {  
    // Display the value of i.  
    trace(i);  
}
```

Discussion

When you want to execute the same action (or slight variations thereof) multiple times within a single frame, use a looping statement to make your code more succinct, easier to read, and easier to update. You can use either a *while* or a *for* statement for this purpose, but generally a *for* statement is a better choice. Both statements achieve the same result, but the *for* statement is more compact and more familiar to most programmers.

The syntax of a *for* statement consists of five basic parts:

The for keyword

Every *for* statement must begin with a *for* keyword.

Initialization expression

Loop typically employs an *index variable* (a *loop counter*) that is initialized when the statement is first encountered. The initialization is performed only once regardless of how many times the loop is repeated.

Test expression

The loop should include a test expression that returns either true or false. The test expression is evaluated once each time through the loop. Generally, the test expression compares the index variable to another value, such as a maximum number of loop iterations. The overall expression must evaluate to true for the *for* statement's body to execute (contrast this with a *do...while* loop, which executes at least once, even if the test expression is false). On the other hand, if the test expression never becomes false, you'll create an infinite loop, resulting in a warning that the Flash Player is running slowly.

Update expression

The update expression usually updates the value of the variable used in the test expression so that, at some point, the conditional test becomes false and the loop ends. The update expression is executed once each time through the loop. An infinite loop is often caused by failing to update the appropriate variable in the update expression (usually the same variable used in the test expression).

Statement body

The statement body is a block of substatements enclosed within curly braces that is executed each time through the loop. If the test expression is never true, the *for* statement's body isn't executed.

The *for* keyword should come first, and it should be followed by the initialization, test, and update expressions enclosed within parentheses. Semicolons must separate the three expressions from one another (although the initialization, test, and update statements are optional, the semicolons are mandatory). The remainder of the *for* loop is made up of the statement body enclosed in curly braces:

```
for (initialization; test; update) {  
    statement body  
}
```

Here is an example of a *for* statement that outputs the numbers from 0 to 999:

```
for (var i:int = 0; i < 1000; i++) {  
    trace(i);  
}  
trace ("That's the end.");
```

To understand the *for* statement, you can follow along with the process the ActionScript interpreter uses to process the command. In the preceding example, the *for* keyword tells the interpreter to perform the statements within the *for* loop as long as the conditional expression is true. The process is as follows:

1. The initialization expression is executed only once, and it sets the variable *i* to 0.
2. Next, the interpreter checks the test expression (*i < 1000*). Because *i* is 0, which is less than 1,000, the expression evaluates to true and the *trace()* action within the *for* statement body is executed.
3. The ActionScript interpreter then executes the update statement, in this case *i++*, which increments *i* by 1.
4. The interpreter then repeats the process from the top of the loop (but skipping the initialization step). So the interpreter again checks whether the test expression is true and, if so, executes the statement body again. It then executes the update statement again.

This process repeats until the test expression is no longer true. The last value displayed in the Output window is 999 because once *i* is incremented to 1,000, the test expression no longer evaluates to true and the loop comes to an end. Once the loop terminates, execution continues with whatever commands follow the loop.

Both the initialization and update expressions can include multiple actions separated by commas. You should not, however, use the *var* keyword more than once in the initialization expression. The following example simultaneously increments *i*, decrements *j*, and displays their values in the Output window:

```
for (var i:int = 0, j:int = 10; i < 10; i++, j--) {
    trace("i is " + i);
    trace("j is " + j);
}
```



The preceding example is not the same as two nested *for* statements (which is shown in the next code block.)

It is also common to use nested *for* statements. When you use a nested *for* statement, use a different index variable than that used in the outer *for* loop. By convention, the outermost *for* loop uses the variable *i*, and the nested *for* loop uses the variable *j*. For example:

```
for (var i:int = 1; i <= 3; i++) {
    for (var j:int = 1; j <= 2; j++) {
        trace(i + " X " + j + " = " + (i * j));
    }
}
```

The preceding example displays the following multiplication table in the Output window:

```
1 X 1 = 1
1 X 2 = 2
2 X 1 = 2
2 X 2 = 4
3 X 1 = 3
3 X 2 = 6
```

It is possible to nest multiple levels of *for* statements. By convention, each additional level of nesting uses the next alphabetical character as the index variable. Therefore, the third level of nested *for* statements typically use *k* as the index variable:

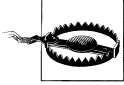
```
for (var i:int = 1; i <= 3; i++) {
    for (var j:int = 1; j <= 3; j++) {
        for (var k:int = 1; k <= 3; k++) {
            trace(i + " X " + j + " X " + k + " = " + (i * j * k));
        }
    }
}
```

Additionally, you can use *for* statements to loop backward or to update the variable in ways other than by simply adding or subtracting one:

```
// Count backward from 10 to 1.
for (var i:int = 10; i > 0; i--) {
    trace(i);
}

// Display a sequence of square roots.
for (var i:Number = 50000; i > 2; i = Math.sqrt(i)) {
    trace(i);
}
```

In this case, the variable *i* winds up holding values other than integers, so it is best to declare it as a *Number* rather than an *int*.



You should not use a *for* statement to perform tasks over time.

Many programmers make the mistake of trying to use *for* statements to animate sprites; for example:

```
for (var i:int = 0; i < 20; i++) {  
    _sprite.x += 10;  
}
```

Whereas the preceding code moves the sprite 200 pixels to the right of its starting point, all the updates take place within the same frame. There are two problems with this. First, the Stage updates only once per frame, so only the last update is shown on the Stage (causing the sprite to jump 200 pixels suddenly rather than moving smoothly in 20 steps). Second, even if the Stage updates often, each iteration through the *for* loop takes only a few milliseconds, so the animation would happen too quickly. For actions that you want to take place over time, use the *enterFrame* event (see *Recipe 1.5*) or a timer (see *Recipe 1.12*).

Moreover, tight repeating loops should not be used to perform lengthy processes (anything that takes more than a fraction of a second). The Flash Player displays a warning whenever a single loop executes for more than 15 seconds. Using the other methods (just mentioned) avoids the warning message and allows Flash to perform other actions in addition to the repeated actions that are part of the loop.

See Also

Recipes *1.5* and *1.12*. The *for* statement is used in many practical situations, and you can see examples in a great many of the recipes throughout this book. See *Recipe 5.2* and *Recipe 12.8* for some practical examples. *Recipe 5.16* discusses *for...in* loops, which are used to enumerate the properties of an object or array.

Repeating a Task over Time

Problem

You want to perform some action or actions over time.

Solution

Use the *Timer* class. Alternatively, listen for the *enterFrame* event of a sprite.

Discussion

The *Timer* class is new to ActionScript 3.0, and is recommended over the earlier *setInterval()* and *setTimeout()* functions. When you create an instance of the *Timer* class, it fires *timer* events at regular intervals. You can specify the delay between events and how many times you want the events to fire in the *Timer* constructor:

```
var timer:Timer = new Timer(delay, repeatCount);
```

You use *addEventListener* to set up a method to handle these events. After you create the timer and set up a listener, use its *start()* method to start it and *stop()* to stop it.

The *Timer* class is part of the *flash.utils* package, and there is also a *TimerEvent* class in the *flash.events* package, so those need to be imported:

```
package {
    import flash.display.Sprite;
    import flash.events.TimerEvent;
    import flash.utils.Timer;

    public class ExampleApplication extends Sprite {
        // Declare and initialize a variable to store the value
        // of the previous timer reading.
        private var _PreviousTime:Number = 0;

        public function ExampleApplication() {
            var tTimer:Timer = new Timer(500, 10);
            tTimer.addEventListener(TimerEvent.TIMER, onTimer);
            tTimer.start();
        }

        private function onTimer(event:TimerEvent):void {
            // Output the difference between the current timer value and
            // its value from the last time the function was called.
            trace(flash.utils.getTimer() - _PreviousTime);
            _PreviousTime = flash.utils.getTimer();
        }
    }
}
```

The *getTimer()* function (previously a top-level function), has been moved to the *flash.utils* package as well. This simply returns the number of milliseconds since the application started.

In the preceding example, even though the interval is theoretically 500 milliseconds in practice its accuracy and granularity depend on computer playback performance in relation to other tasks demanded of the processor. There are two implications to this:

- Don't rely on timers to be extremely precise.
- Don't rely on timer intervals to be smaller than approximately 10 milliseconds.

If you want to emulate the functionality of the *setInterval()* function, set the repeat count to zero. This causes the timer event to fire indefinitely. In this case, the *stop()* method is analogous to the *clearInterval()* function, and stops the timer from firing further events.

Similarly, if you want to duplicate the *setTimeout()* function, set the repeat count to one. The timer waits the specified amount of time, fires one event, and ends.

One of the neat things you can do with the *Timer* class is create animations that are independent of the movie's frame rate. With a timer you can call a method at any interval you want. Here is an example in which two timers are set—one for a square sprite (every 50 milliseconds)—and one for a circle sprite (every 100 milliseconds):

```
package {
    import flash.display.Sprite;
    import flash.events.TimerEvent;
```

```

import flash.utils.Timer;

public class ExampleApplication extends Sprite {
    private var _square:Sprite;
    private var _circle:Sprite;

    public function ExampleApplication() {
        // Create the two sprites and draw their shapes
        _square = new Sprite();
        _square.graphics.beginFill(0xff0000);
        _square.graphics.drawRect(0, 0, 100, 100);
        _square.graphics.endFill();
        addChild(_square);
        _square.x = 100;
        _square.y = 50;

        _circle = new Sprite();
        _circle.graphics.beginFill(0x0000ff);
        _circle.graphics.drawCircle(50, 50, 50);
        _circle.graphics.endFill();
        addChild(_circle);
        _circle.x = 100;
        _circle.y = 200;

        // Create the two timers and start them
        var squareTimer:Timer = new Timer(50, 0);
        squareTimer.addEventListener(TimerEvent.TIMER, onSquareTimer);
        squareTimer.start();

        var circleTimer:Timer = new Timer(100, 0);
        circleTimer.addEventListener(TimerEvent.TIMER, onCircleTimer);
        circleTimer.start();
    }

    // Define the two handler methods
    private function onSquareTimer(event:TimerEvent):void {
        _square.x++;
    }

    private function onCircleTimer(event:TimerEvent):void {
        _circle.x++;
    }
}

```

It is also possible to use the *enterFrame* event of a sprite to have some action (or actions) repeat over time. The *Timer* technique offers some advantages over the *enterFrame* event method, most notably that it allows you to create intervals that differ from the frame rate of the *.swf*. With *enterFrame*, the handling method is called at the frame rate.

With that said, there are still times when using *enterFrame* is appropriate. For example, you may want something to occur at the frame rate of the *.swf*. One such scenario is when you want to reverse the playback of the frames in a movie clip.

Creating Reusable Code

Problem

You want to perform a series of actions at various times without duplicating code unnecessarily throughout your movie.

Solution

Create a method and then call (i.e., invoke) it by name whenever you need to execute those actions. When a function is a member of a class, it is often called a *method*.

Here is how to create a *method* of a class:

```
    accessModifier function functionName ( ): ReturnDataType {  
    // Statements go here.  
}
```

To *call* (i.e., execute) the named method, refer to it by name, such as:

```
functionName( );
```

Discussion

Grouping statements into a method allows you to define the method once but execute it as many times as you'd like. This is useful when you need to perform similar actions at various times without duplicating the same code in multiple places. Keeping your code centralized in methods makes it easier to understand (because you can write the method once and then ignore the details when using it) and easier to maintain (because you can make changes in one place rather than in multiple places).

Like class variables, methods can be declared with *access modifiers*. These determine which other classes are able to call the methods. The available access modifiers are:

private

Can only be accessed from within the class itself.

protected

Can be accessed by the class or any subclass. This is instance-based. In other words, an instance of a class can access its own protected members or those of its superclasses. It cannot access protected members on other instances of the same class.

internal

Can be accessed by the class or any class within the same package.

public

Can be accessed by any class.

The definition of *private* has changed since ActionScript 2.0, where it allowed access by subclasses. If you do not specify an access modifier explicitly, the method takes on the default *internal* access.

The following class defines a *drawLine* method and calls it 10 times, rather than repeating the three lines of drawing code for each line:

```

package {
    import flash.display.Sprite;

    public class ExampleApplication extends Sprite
    {
        public function ExampleApplication() {
            for(var i:int=0;i<10;i++) {
                drawLine();
            }
        }

        private function drawLine():void {
            graphics.lineStyle(1, Math.random() * 0xffffff, 1);
            graphics.moveTo(Math.random() * 400, Math.random() * 400);
            graphics.lineTo(Math.random() * 400, Math.random() * 400);
        }
    }
}

```

Another important method type is a *static* method. Static methods aren't available as a member of an instance of that class, but instead are called directly from the class itself. For example, in a class named `ExampleApplication`, you could define a static method as follows:

```

public static function showMessage():void {
    trace("Hello world");
}

```

You could then call that method like so:

```

ExampleApplication.showMessage();

```

Some classes contain nothing but static methods. The *Math* class is an example. Note that you don't have to create a new *Math* object to use its methods; you simply call the methods as properties of the class itself, such as *Math.random()*, *Math.round()*, and so on.

Generalizing a Method to Enhance Reusability

Problem

You want to perform slight variations of an action without having to duplicate multiple lines of code to accommodate minor differences.

Solution

Add parameters to your method to make it flexible enough to perform slightly different actions when invoked rather than performing exactly the same action or producing the same result each time.

Define the parameters that account for the variability in what you want the method to do:

```

private function average (a:Number, b:Number, c:Number):void {
    trace("The average is " + (c + b + a)/3);
}

```

If you don't know the exact number of parameters the method will receive, use the built-in `arguments` array to handle a variable number of parameters.

Discussion

A method that doesn't accept parameters generally does exactly the same result each time it is invoked. However, you will often need to perform almost exactly the same actions as an existing method, but with minor variations. Duplicating the entire method and then making minor changes to the second version is a bad idea in most cases. Usually, it makes your code harder to maintain and understand. More importantly, you'll usually find that you need not only two variations but many variations of the method. It can be unnecessarily difficult to maintain five or six variations of what should ideally be wrapped into a single method. The trick is to create a single method that can accept different values to operate on.

For example, let's say you have an `average()` method for averaging a set of numbers. Instead of having it always average the same two numbers, you want to specify arbitrary values to be averaged each time it is invoked. This can be accomplished with parameters.

The most common way to work with parameters is to list them within the parentheses in the method declaration. The parameter names should be separated by commas, and when you invoke the method you should pass it a comma-delimited list of arguments that corresponds to the parameters it expects.



The terms “parameters” and “arguments” are often used interchangeably to refer to the variables defined in the method declaration or the values that are passed to a method when it is invoked.

The following is a simple example of a method declaration using parameters:

```
// Define the function such that it expects two parameters: a and b.
private function average(a:Number, b:Number):Number {
    return (a + b)/2;
}
```

Now here's a method invocation in which arguments are passed during the method call:

```
// When you invoke the function, pass it two arguments, such as
// 5 and 11, which correspond to the a and b parameters.
var averageValue:Number = average(5, 11);
```

In most situations it is best to declare the parameters that the method should expect. However, there are some scenarios in which the number of parameters is unknown. For example, if you want the `average()` method to average any number of values, you can use the built-in `arguments` array that is available within any function's body. All the parameters that are passed to a function are automatically placed into that function's `arguments` array.

```
// There is no need to specify the parameters to accept when using the
// arguments array.
private function average():Number {
    var sum:Number = 0;

    // Loop through each of the elements of the arguments array, and
    // add that value to sum
    for (var i:int = 0; i < arguments.length; i++) {
```

```

        sum += arguments[i];
    }
    // Then divide by the total number of arguments
    return sum/arguments.length;
}

// You can invoke average() with any number of parameters.
var average:Number = average (1, 2, 5, 10, 8, 20);

```



Technically, `arguments` is an object with additional properties beyond that of a basic array. However, while `arguments` is a special kind of array, you can still work with it in the same ways that you would a regular array.

Exiting a Method

Problem

You want to exit a method.

Solution

Methods terminate automatically after the last statement within the method executes. Use a *return* statement to exit a method before reaching its end.

Discussion

The *return* statement exits the current method and the ActionScript interpreter continues execution of the code that initially invoked the method. Any statements within the method body that follow a *return* statement are ignored.

```

private function sampleFunction ():void {
    return;
    trace("Never called");
}

```

```

// Called from within another method:
sampleFunction();
// Execution continues here after returning from the sampleFunction() invocation

```

In the preceding example, the *return* statement causes the method to terminate before performing any actions, so it isn't a very useful method. More commonly, you will use a *return* statement to exit a method under certain conditions. This example exits the method if the password is wrong:

```

private function checkPassword (password:String):void {

    // If password is not "SimonSays", exit the function.
    if (password != "SimonSays") {
        return;
    }

    // Otherwise, perform the rest of the actions.
    showForm ("TreasureMap");
}

```

```

}

// This method call uses the wrong password, so the
// function exits.
checkPassword("MotherMayI");

// This method call uses the correct password, so the function
// shows the TreasureMap form.
checkPassword("Si monSays");

```

In the preceding example, you may notice that the method is declared as `void`, yet it is possible to use a *return* statement within the method without getting a compiler error. When a *return* statement is used simply to exit from a method, it is valid within a method declared as `void`.



In ActionScript 2.0, the function was `Void`. In ActionScript 3.0, it is lowercase `void`.

However, if you attempt to actually return a value in such a method, the compiler generates an error.

```

private function sampleMethod():void {
    return "some value"; // This causes the compiler to generate an error.
}

```

Obtaining the Result of a Method

Problem

You want to perform some method and return the results to the statement that invoked the function.

Solution

Use a *return* statement that specifies the value to return.

Discussion

When used without any parameters, the *return* statement simply terminates a method. However, any value specified after the *return* keyword is returned to statement that invoked the method. Usually, the returned value is stored in a variable for later use. The datatype of the return value must match the return type of the method:

```

private function average (a:Number, b:Number):Number {
    return (a + b)/2;
}

```

Now we can call the *average()* method and store the result in a variable and use the result in some way.

```

var playerScore:Number = average(6, 10);
trace("The player's average score is " + playerScore);

```

You can use the return value of a method, without storing it in a variable, by passing it as a parameter to another function, such as:

```
trace("The player's average score is " + average(6, 10));
```

Note, however, that if you do nothing with the return value of the function, the result is effectively lost. For example, this statement has no detectable benefit because the result is never displayed or used in any way:

```
average(6, 10);
```

Handling Errors

Problem

You want to programmatically detect when certain errors occur and handle them using code.

Solution

Use a *throw* statement to throw an error when it is detected. Place any potentially error-generating code within a *try* block, and then have one or more corresponding *catch* blocks to handle possible errors.

Description

Flash Player 8.5 supports a *try/catch* methodology for handling errors in ActionScript. That means you can write code that can intelligently deal with certain error types should they occur. While you cannot handle syntax errors (the *.swf* won't even compile in that case), you can handle most other error types, such as missing or invalid data. The benefit is that you can attempt to resolve the situation programmatically.

An example may help to illustrate when and how you might use *try/catch* methodology: Consider an application that draws a rectangle based on user-input dimensions. To draw a rectangle within the application, you want to have certain range limitations on the dimensions the user can input. For example, you may want to make sure the values are defined, valid numeric values greater than 1 and less than 200. While there are certainly ways you can work to ensure the quality and validity of the data before even attempting to draw the rectangle, you can also use *try/catch* methodology as a fail-safe. You can have Flash attempt to draw the rectangle, but if the dimension values are detected to be invalid or out of range, you can throw an error that can be handled programmatically. At that point you can do many things, from simply skipping the action, to substituting default data, to alerting the user to enter valid data.

There are two basic parts involved in working with errors in ActionScript: throwing the error and catching the error. There are several errors, which are thrown automatically by the player, such as *IllegalOperationError*, *MemoryError*, and *ScriptTimeoutError*. These are in the *flash.errors* package. But you can also detect when an error has occurred and throw your own custom error. You can throw an error using the *throw* statement. The *throw* statement uses the *throw* keyword followed by a value or reference that should be thrown. Most frequently you should throw an *Error* object or an instance of an *Error* subclass. For example:

```
throw new Error("A general error occurred.");
```

As you can see, the *Error* constructor accepts one parameter, a message to associate with the error. The parameter is optional, and depending on how you are handling the errors, you may or may not choose to use it. However, in most cases it makes sense to specify an error message. It is possible, then, to log the error messages for debugging purposes.

Once an error has been thrown, Flash halts the current process and looks for a *catch* block to handle the error. This is where the *try* and *catch* blocks come into play. Any code that could potentially throw an error should be enclosed in a *try* block. Then, if an error is thrown, only the code in the *try* block is halted, and the associated *catch* block is called. The following is the simplest scenario:

```
try {
    trace("This code is about to throw an error.");
    throw new Error("A general error occurred.");
    trace("This line won't run");
}
catch (errObject:Error) {
    trace("The catch block has been called.");
    trace("The message is: " + errObject.message);
}
```

The preceding code produces the following in the Output panel:

```
This code is about to throw an error.
The catch block has been called.
The message is: A general error occurred.
```

Of course, the preceding example is overly simplistic, and you wouldn't realistically use code in an actual application, but it does illustrate the basic process. You can see that as soon as the error is thrown, the *try* block is exited, and the *catch* block is run and passed a reference to the *Error* object that was thrown.

Much more frequently, the error is thrown from within a function or method. Then Flash looks to see if the *throw* statement within the function is contained within a *try* block. If so, it calls the associated *catch* block as you've seen already. However, if the *throw* statement in the function is not within a *try* block, Flash exits the function and next looks to see if the function call was made within a *try* block. If so, it halts the code in the *try* block and runs the associated *catch* block. Again, a very simple example:

```
private function displayMessage(message:String):void {
    if(message == undefined) {
        throw new Error("No message was defined.");
    }
    trace(message);
}

try {
    trace("This code is about to throw an error.");
    displayMessage( );
    trace("This line won't run");
}
catch (errObject:Error) {
    trace("The catch block has been called.");
    trace("The message is: " + errObject.message);
}
```

In the preceding example the Output panel would display the following:

This code is about to throw an error.
The catch block has been called.
The message is: No message was defined.

As you can see from the output, the code works very similarly to the way in which the previous example worked, except the throw statement is hidden within a function instead of being called directly within the *try* block. The advantage is that you can start to then create functions and methods that are intelligent enough to know if and when to throw errors. You can then simply use those functions and methods within *try* blocks, and you can handle any errors should they occur.

The following code illustrates a more realistic example:

```
// Define a function that draws a rectangle within a specified sprite
private function drawRectangle(sprite:Sprite, newWidth:Number, newHeight:Number):void {

    // Check to see if either of the specified dimensions are not
    // a number. If so, then thrown an error.
    if(isNaN(newWidth) || isNaN(newHeight)) {
        throw new Error("Invalid dimensions specified.");
    }

    // If no error was thrown, then draw the rectangle.
    sprite.graphics.lineStyle(1, 0, 1);
    sprite.graphics.lineTo(newWidth, 0);
    sprite.graphics.lineTo(newWidth, newHeight);
    sprite.graphics.lineTo(0, newHeight);
    sprite.graphics.lineTo(0, 0);
}
```

Now we can call the *drawRectangle()* method using a *try/catch* statement.

```
try {

    // Attempt to draw two rectangles within the current sprite.
    // In this example it is assumed that the variables for the dimensions
    // are retrieving values from user input, a database, an XML file,
    // or some other datasource.
    drawRectangle(this, widthA, heightA);
    drawRectangle(this, widthB, heightB);
}
catch(errObject:Error) {

    // If an error occurs, clear any rectangles that were drawn from
    // the sprite. Then display a message to the user.
    this.graphics.clear();
    tOutput.text = "An error occurred: " + errObject.message;
}
```

In addition to the *try* and *catch* blocks, you can also specify a *finally* block. The *finally* block contains code that is called regardless of whether an error was thrown. In many cases the *finally* block may not be necessary. For example, the following two examples do the same thing:

```
//Without using finally:
private function displayMessage(message:String):void {
    try {
        if(message == undefined) {
```

```

        throw new Error("The message is undefined.");
    }
    trace(message);
}
catch (errObject:Error) {
    trace(errObject.message);
}
trace("This is the last line displayed.");
}
//With finally:
private function displayMessage(message:String):void {
    try {
        if(message == undefined) {
            throw new Error("The message is undefined.");
        }
        trace(message);
    }
    catch (errObject:Error) {
        trace(errObject.message);
    }
    finally {
        trace("This is the last line displayed.");
    }
}

```

However, the *finally* block runs no matter what occurs within the *try* and *catch* blocks, including a *return* statement. So the following two functions are not the equivalent:

```

//Without using finally:
private function displayMessage(message:String):void {
    try {
        if(message == undefined) {
            throw new Error("The message is undefined.");
        }
        trace(message);
    }
    catch (errObject:Error) {
        trace(errObject.message);
        return;
    }
    // This line won't run if an error is caught.
    trace("This is the last line displayed.");
}
//With finally:
private function displayMessage(message:String):void {
    try {
        if(message == undefined) {
            throw new Error("The message is undefined.");
        }
        trace(message);
    }
    catch (errObject:Error) {
        trace(errObject.message);
        return;
    }
}

```

```
finally {  
    // This runs, even if an error is caught.  
    trace("This is the last line displayed.");  
}  
}
```

You can create much more complex error handling systems than what is shown in this recipe. Throughout this book you will find examples of more complex error handling in appropriate contexts.

Index

!
! logical NOT, 23
!= (logical inequality operator), 16, 24
!== (inequality operator), 17

#include directives, 6

&
&& logical AND, 23

(
() (parentheses)
conditional operators and, 24
(semicolon), 8

)
) , 8

*
*= (compound assignment operator), 14

+
++ increment operator, 15
+=
compound assignment operator, 14

-
-- decrement operator, 15
-= (compound assignment operator), 14

.
.as files, 4
placing code, 6
.fla files, 4
.html files, 4
.swf files, 4

/
/**/ comments, 8
// (comments), 8
/= (compound assignment operator), 14

<
< (less than)
equality operators and, 18
<= operator, 18

=
= (equal sign)

assignment statements and, 14
differences with == and, 17
variables, assigning, 7
=== equality operator, 17

>
> (greater than)
equality operators and, 18
>= operator, 18

?
?: (ternary conditional operator), 19

A
access modifiers, 31
actions (statements), 8
ActionScript interpreter, 8
ActionScript Project Wizard, 4
addEventListener() method, 11
repeating tasks and, 29
AND (&&) logical, 23
applications
properties, customizing, 4
arguments
command-line, 4
methods, 33
assignment operators, 7, 14
equality operators, differences and, 17

B
basics, 3
bin folder, creating projects and, 4
Booleans
conditional operators and, 24
equality expressions returning, 16
break statements, 21

C
case keyword, 21
catch blocks, 36
classes, 8
placing code, 6
properties, customizing, 4
variables, 7
code
placing, 6
reusable, 31, 32
command-line compilers, 5
commands, 8
compilers, 5
composite datatypes, 18
compound assignment operators, 14
conditional statements, 19
Console view, using trace, 9
constructors, 7

- placing code, 6
- curly braces ({ }), 6
 - for statements and, 26

D

- datatypes, 18
- default keyword, 22
- Delegate class, 12
- directories
 - holding classes, 6
- drawRectangle() method, 38

E

- Eclipse IDE, 3
 - parameters, keeping track of, 5
 - trace, using, 10
- else statements, 20
- else...if statements, 20
- enterFrame event, 11, 28
 - repeating tasks over time, 28
 - responding to mouse/key events, 12
- equal sign (=)
 - assignment statements and, 14
 - differences with == and, 17
 - variables, assigning, 7
- equality operators, 17
- equality, checking, 16
- error handling, 36
- ErrorReportingEnable variable (mm.cfg), 10
- event object, 11
- EventDispatcher class, handling events, 11
- events, 8
 - handling, 10

F

- finally blocks, 38
- flash.display package, 7
- flash.errors package, 36
- flash.utils package, 29
- Flex Builder, 3
- for keyword, 25
- for statement, 23
 - repeating operations many times, 25
- fps (frames per second), 5
- frames per second (fps), 5
- functions, 8

G

- getTimer() function, 29
- greater than (>)
 - equality operators and, 18

H

- handlers, 8
- handling errors, 36

I

- if statements, 19
 - complex conditional testing, using, 23
- IllegalOperationError, 36
- import statements, 6
- index variables (loop counters), 25
- inequality, checking, 16

- initialization expressions, 25
- instances (object), 8
- int keyword, 27
- internal keyword, 7
 - reusing code and, 31
- internal variables, 8
- interpreter (ActionScript), 8
- isNaN(), 16

K

- key events, responding to, 12
- KeyboardEvent class, 13
- keyDown event, 13

L

- less than (<)
 - equality operators and, 18
- logical AND (&&), 23
- logical inequality operator (!=), 16, 24
- logical NOT (!), 23
- logical OR (||), 23
- loop counters, 25
- loop statements, 25

M

- Mac OS X mm.cfg file, locating, 10
- Macromedia XML (MXML), 3
- Math class, 32
- mathematical operators, 14
- MaxWarnings variable (mm.cfg), 10
- MemoryError, 36
- messages, tracing, 9
- metadata, specifying properties, 4
- methods, 8
 - exiting, 34
 - generating to enhance reusability, 32
 - obtaining results, 35
 - placing code, 6
 - reusing code, 31
- mm.cfg file, 10
- mouse events
 - responding to, 12
- MouseEvent class, 13
- MXML (Macromedia XML), 3
- mxmmlc
 - viewing command-line arguments, 5

N

- NaN (not a number) constant, 17
- Navigator view (Flex Builder 2), 4
 - nested for statements, 27
- New ActionScript Project Wizard, 4
- NOT (!) logical, 23
- not a number (NaN) constant, 17
- Number keyword, 27

O

- objects, 8
- onMouseDown method, 13
- onMouseUp method, 13
- OR (||) logical, 23
- Outline view (Flex Builder 2), 4

P

- package keyword, 6
- parameters (method), generalizing for enhancing reusability, 32
- parentheses (())
 - conditional operators and, 24
- postfix operators, 15
- prefix operators, 15
- primitive datatypes, 18
- private keyword, 7
 - reusing code, 31
- Project Wizard, 4
- projects (Flex), 3
 - creating, 3
- properties, 7
 - applications, customizing, 4
- protected keyword, 7
 - reusing code, 31
- protected variables, 8
- public keyword, 7
 - reusing code and, 31

R

- reserved words, 7
- return statements, 34
- reusable code, 31, 32

S

- scope, 8
- ScriptTimeoutError, 36
- semicolon (\), 8
- setInterval() function, 28
- setTimeout() function, 28
- Sprite class, 7
- start() method (addEventListener), 29
- statement body, 26
- statements, 8
- static methods, 32
- stop() (addEventListener), 29
- strict equality/inequality, 16
- strict flag, 16
- StringUtils class, 7

- switch statements, 19

T

- ternary conditional operator (? :), 19
- test expressions, 25
- Timer class, 28
- trace(), 9
- TraceOutputFileEnable variable (mm.cfg), 10
- TraceOutputFileName variable (mm.cfg), 10
- try blocks, 36
- type parameter of addEventListener(), 11

U

- update expressions, 26

V

- var keyword, 7
- variables, 7
- void, declaring methods, 35

W

- while statement, 23
- Windows 2000
 - mm.cfg file, locating, 10
- Windows XP
 - mm.cfg file, locating, 10
- Wizard (Project), 4

- \
- \\, 8

- {
- { } (curly braces), 6
 - for statements and, 26

- |
- || logical OR, 23