# ActionScript 3.0 Coding Standards

*Nate Chatellier*

## *Why Use Standards?*

As a programmer, attention to detail, consistency, and thoroughness are critical attributes. Coding standards (aka “coding conventions” or “coding style”) aid and develop these abilities. They make it easier to recognize when something is out of place or incomplete.

Additionally, standards simplify and strengthen team development. If everyone on the team writes with the same style, it can greatly reduce the time it takes to understand each other's code. If multiple styles exist within a given project, class, or even a single function, a great amount of time might be wasted interpreting the style, before one can even attempt to understand the functionality.

Finally, good standards and naming conventions encourage code that is well thought out and self-documented.

## *Which Style is Correct?*

It should be noted that there aren't really correct or incorrect styles. The important thing is that the team agrees to comply with a single set of standards, setting aside personal bias for the greater good of the team.

Having said this, there is also no reason to reinvent the wheel or go against the grain. Most languages or industries have standards published by the big players within that industry. For example:

- Microsoft (MSDN Coding Style Conventions)
- Apple (Coding Guidelines for Cocoa)
- Google (C++ Style Guide)
- Adobe (AS3 & Flex Coding Conventions)

## *This Document*

The purpose of this document is to suggest a simplified set of standards to be followed for ActionScript 3.0 (AS3) programming. The majority of the standards are directly taken from Adobe's Coding Conventions (link above), with simplification in certain areas and expansion in others.

# General

## Tab Stops

The commonly accepted AS3 standard is to use tabs instead of spaces with 4 character tabstop.

## Brace Placement

Adobe's standard is to place all `{` or `}` on a line by themselves in every situation, aligning the braces with the code block.

*Example*:

```
public function someFunction($nJ:uint):void
{
    for (var i:uint = 0; i < 10; i++)
    {
        if (i < $nJ)
        {
            $nJ--;
        }
    }
}
```

# Variables

## Variable Prefixes

Variables should be defined using Hungarian Notation. That is, their name contains a short prefix denoting the variable's type. Following is a table of the most common prefixes:

| Prefix | Variable Type |
|---|---|
| s | `String` |
| n | `Number`, `int`, or `uint` |
| b | `Boolean` |
| a | `Array` |
| v | `Vector` |
| t | `Timer` |
| p | `Point` |
| r | `Rectangle` |
| x | Any `XML` object |
| o | `Object` |
| mc | `MovieClip`, `Sprite`, or anything that extends from them. |
| bt | `SimpleButton` or anything that extends from it. |
| txt | `TextField` or anything that extends from it. |
| tf | `TextFormat` or anything that extends from it. |

*Example*:

```
var mcSomeSprite        :Sprite;
var nSomeNumber         :uint;
```

### Class Level Variables

All non-public, non-constant, class level variables should be prefixed with an underscore ("_"). When accessing a public class level variable, always use the `this` keyword so that the variable can still be recognized as a class level variable rather than a local one at a glance.

Also, whenever possible, class level variables should be instantiated as part of the declaration, not in the constructor.

*Example*:

```
public var            mcSomeSprite   :Sprite;
protected var         _pSomePoint    :Point = new Point(0, 0);
```

### Parameter Variables

Function parameters should be prefixed with a `$` character so they can be recognized at a glance.

*Example*:

```
public function doSomething($sParam1:String, $sParam2:String):void
```

### Naming

Use meaningful variable names and avoid abbreviations unless they are very commonly used. Readability should always be favored over fewer keystrokes. For example, instead of using the variable name `ew`, use the much more meaningful name of `mcEnemyWeapon`.

### Event Names

Event names should be defined as string constants with the class name preceding the event description.

*Example*:

```
public static const   ENEMY_DESTROYED     :String  = "ThisClassName.enemyDestroyed";
public static const   HEALTH_RESTORED     :String  = "ThisClassName.healthRestored";
```

### Casting

Whenever a variable's type is ambiguous to the compiler (and polymorphism is not being used), cast it. For example, if manipulating the width of a Sprite stored in an Array, the following code should be used:

*Do this*:

```
Sprite(aSomeArray[0]).width = 50;
```

*Not this*:

```
aSomeArray[0].width = 50;
```

This increases execution speed, helps the compiler capture type errors, makes your code more readable, and allows proper code hinting in many IDE's.

## *Class Structure*

### Imports

Imports should be logically grouped by their most significant package.

*Example*:
```
import flash.display.Sprite;
import flash.display.SimpleButton;
import com.natejc.utils.StageRef;
import com.greensock.TweenMax;
import com.greensock.easing.Quad;
```

Also, Adobe's standard is to be specific with imports and avoid wild cards. Plus, wild cards can prevent code hinting altogether in many IDE's.

*Do this*:
```
import flash.display.Sprite;
import flash.display.SimpleButton;
```

*Not this*:
```
import flash.display.*;
```

### Constructors

Adobe's standard is that variables should be instantiated where they are defined, not in the constructor. The one exception is for non-base class constructors altering the value of base class variables.

### Overridden Functions

Adobe's standard is that the `override` keyword should appear before the rest of a function definition.

*Do this*:
```
override public function doStuff():void
```
*Not this*:
```
public override function doStuff():void
```

### Function Separators

For the sake of code separation and readability, use the following divider, preceded and proceeded by a single line break, to separate function definitions:

```
/* ------------------------------------------------------------------------------------ */
```

### Destroy Function

Custom objects that will ever need to be removed from memory after construction, should have a public `destroy` function, which relinquishes all memory and cpu cycles claimed by that object. This destructor function, if it exists, should always immediately follow the constructor.

## *Formatting*

### if statements

The `if` keyword should be followed by a single space, followed by the left parenthesis. Also, the braces may be omitted if only a single line of execution proceeds the `if` statement.

```
if (bTestCondition)
     txtSample.text = "true";

if (bTestCondition)
{
     txtSample.text = "true";
     txtSample.y    = 100;
}


if (nValue < 10)
{
     txtSample.text = "Less than 10";
     txtSample.y    = 10;
}
else if (nValue < 20)
{
     txtSample.text = "Less than 20";
     txtSample.y    = 20;
}
else
{
     txtSample.text = "Too big!";
     txtSample.y    = 100;
}
```

### for statements

The `for` keyword should be followed by a single space, followed by the left parenthesis. Also, the braces may be omitted if only a single line of execution proceeds the `for` statement.

```
   for (var i:uint = 0; i < 10; i++)
      mcSprite.width += 10;

   for (var i:uint = 0; i < 10; i++)
   {
      mcSprite.width += 10;
      mcSprite.height += 10;
   }
```

### switch statements

Adobe's standard for `switch` statements is as follows:

```
switch (n)
{
    case 1:
    {
        a = foo();
        break;
    }

    default:
    {
        a = blah();
        break;
    }
}
```

### while and do while statements

The `while` keyword should be followed by a single space, followed by the left parenthesis. Also, the braces may be omitted if only a single line of execution proceeds the `while` statement.

```
while (i < n)
   doSomething(i);

while (i < n)
{
   doSomething(i);
   doSomethingElse(i);
}

do
{
   doSomething(i);
} while (i < n);
```

### try catch finally statements

```
try
{
    mcSprite.addChild(mcSprite);
}
catch (err:ArgumentError)
{
    trace(err); // output: ArgumentError: Error #2024
}
finally
{
   this.addChild(mcSprite);
}
```

### Operators and Assignments

Put single spaces around assignment, comparison, and infix operators:

*Correct Examples*:

```
a = b
a = b + c
a == b
a < b
```

Don't put any spaces between prefix or postfix operators and their operand:

*Correct Examples*:

```
!a
i++
++j
```

## *Commenting*

### ASDoc

All classes, public class properties, and class functions must have ASDoc compatible comments. All ASDoc comments should be written in complete sentences (including the period at the end). See the following link for more information on ASDoc:

http://livedocs.adobe.com/flex/201/html/wwhelp/wwhimpl/common/html/wwhelp.htm?context=LiveDocs_Book_Parts&file=asdoc_127_1.html

### TODO Statements

Use the common TODO comment to denote things that still need to be done. Include your initials at the end of the statement so others know who to visit should they have further questions.

*Example*:

```
// TODO: finish writing this function. ~NJC
```

### General Comments

Use commenting to help explain very complex procedures or to organize complex code blocks into sections. However, if good naming conventions are used, most smaller code blocks should be self documented.

## Full Class Example

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

    /**
     * Class purpose.
     *
     * @author Nate Chatellier
     */
    public class MySampleClass extends Sprite
    {
        /** ASDoc formatted description of this variable. */
        public var    mcMySprite  :Sprite;

        /** ASDoc formatted description of this variable. */
        protected var _bMyBoolean :Boolean; //

        /* ------------------------------------------------------------------------------------------ */

        /**
         * Constructs the MySampleClass object.
         */
        public function MySampleClass()
        {
            reset(false);
        }

        /* ------------------------------------------------------------------------------------------ */

        /**
         * Relinquishes all memory and cpu cycles used by this object.
         */
        public function destroy():void
        {
            // do stuff
        }

        /* ------------------------------------------------------------------------------------------ */

        /**
         * Reset this class and this class's children to their initial states.
         *
         * @param  bResetChildrenFalse if this class's children should not also be reset.
         */
        public function reset($bResetChildren:Boolean = true):void
        {
            _bMyBoolean = false;

            if ($bResetChildren)
            {
                // do some stuff
            }
        }

        /* ------------------------------------------------------------------------------------------ */
    }
}
```