Rivello Multimedia Consulting
Post Comments to: http://www.blog.rivello.org/?p=159

Best Practices for

Coding ActionScript 3.0 & MXML

v1.1

1. INTRODUCTION

This document aims to establish coding guidelines to applications written with Adobe Flex 3 and ActionScript 3.

Establish a coding convention matters in this context because the majority of the time in the software development life cycle is spent doing maintenance. Because of this, helping the comprehension of code passages is a must, considering that the person who's going to perform the maintenance will not always be the same person who built it in the first place. With a common language, developers can rapidly understand other people's code. Besides, the application or components code can be distributed or sold to third parties.

The premises of coding conventions are:

- Consistency
- Code comprehension

The practices established in this document are based on
Java coding conventions and conventions seen at Adobe Flex 3 SDK.

2. FILES

2.1. Files & File Suffixes

- MXML code: .mxml
  - Best practice dictates that this should contain mostly MXML. ActionScript 3.0 is to be externalized to an ActionScript 3.0 file.
- ActionScript code: .as
  - ActionScript 3.0 Classes – Best practice dictates that this is most common type of file in any project.
  - ActionScript 3.0 Files – Best practice dictates that use of these files is to be avoided. While a common solution to externalize code from an MXML file, an ActionScript 3.0 class is to be used instead.
- CSS code: .css , .swf
  - Best practice dictates to use .css for compile-time CSS and .swf for run-time CSS. Run-time CSS offers more flexibility in application development but requires more preparation. See Adobe Flex 3 Online Help for more information (http://livedocs.adobe.com/flex/3/html/help.html?content=styles_10.html).

2.2. File names

- Must not contain spaces, punctuation marks, or special characters
- ActionScript
  - Classes and Interfaces use UpperCamelCase
  - Interfaces always start with an upper case I
    - IUpperCamelCase
  - Includes use lowerCamelCase
  - Namespace definitions use lower case.
- MXML
  - Always use UpperCamelCase.
- CSS
  - Always use lowerCamelCase.

2.3. Encoding

- All files must be in UTF8 format.

3. ACTIONSCRIPT 3.0

3.1. File Organization

An ActionScript class must contain the following structure:

#	Element	Description
1	Initial Comment	
2	Package definition	
3	Namespace declaration • If it has one, it is the last section	A file that defines a namespace only does that.
4	*Import* statements 1. Package flash 2. Package mx 3. Package com.adobe 4. Packages of third party in alphabetical order 5. Package of the project this files belongs to Use fully qualified imports, i.e., without the asterisk. • Prefer: import mx.core.Application; • Avoid: import mx.core.*;	Inside these sections, all imports must in alphabetical order. If there's a namespace import, this must precede the class imports of the same package.
5	*use* declarations (namespace)	In alphabetical order.
6	Metadata 1. Event 2. Style 3. Effect 4. Other metadata in alphabetical order.	
7	Class or interface definition	
8	Variables 1. getter/setter 2. public a. regular b. static c. constant 3. private a. regular b. static c. constant 4. protected a. regular b. static	

	c. constant 5. custom namespaces 6. a. In alphabetical order	
9	Instance variables aren't handled by getters and setters 1. public 2. internal	
	3. protected 4. private 6. custom namespaces a. In alphabetical order	
10	Constructor	

11	Getters and setter managed variables and the get and set methods themselves, as related variables. Example:
	
	public function get sample () : String {
	return _sample_str;
	}
	public function set sample (aValue : String) : void {
	_sample_str = aValue;
	}
	private var _sample_str : String;
		See the variables section on this document for rules about variables managed by get and set methods.
		
		Typically only three lines are used for the complete getter/setter, unlike example shown here, which lacks the margins to do so.

12	Methods	Grouped by functionality, not by scope.

13	Events
	
	//Event Dispatchers
	private function dispatchSample () : void
	{
	
	//dispatchEvent(new Event (Event.SAMPLE));
	}
	
	//Event Handlers
		Grouped first by 'event dispatchers,' then by 'event handlers.' Event dispatchers are methods really that the class calls internally. This way

	private function onInit (aEvent : Event) : void
	{
	
	...
	
	}
		dispatchEvent is not called sporadically throughout code, only in this section for easy readability.

Example: TemplateClass.as

```
//Marks the right margin of code
//////////////////////////////////////////////////////////////////////
package com.company.templates
{
	//---------------------------------------
	//  Imports
	//---------------------------------------
	import flash.events.Event;
	import flash.events.EventDispatcher;


	//---------------------------------------
	//  Class
	//---------------------------------------
	/**
	 * This is the typical format of a simple multiline comment
	 * such as for a <code>TemplateClass<code> class.
	 *
	 * <p><u>REVISIONS</u>:<br>
	 * <table width="500" cellpadding="0">
	 * <tr><th>Date</th><th>Author</th><th>Description</th></tr>
	 * <tr><td>MM/DD/YYYY</td><td>AUTHOR</td><td>Class created.</td></tr>
	 * <tr><td>MM/DD/YYYY</td><td>AUTHOR</td><td>DESCRIPTION.</td></tr>
	 * </table>
	 * </p>
	 *
	 * @example Here is a code example.
	* <listing version="3.0" >
	 *    //Code example goes here.
	 * </listing>
	 *
	* <span class="hide">Any hidden comments go here.</span>
	*
	 */
	public class TemplateClass extends EventDispatcher implements
				ITemplateInterface
	{

		//---------------------------------------
		//  Properties
		//---------------------------------------
		//PUBLIC GETTER/SETTERS
		/**
		 * This is the typical format of a simple comment for sample.
		 *
		 */
		public function get sample () : String { return _sample_str; }
```

```
public function set sample (aValue : String) : void {
                                    _sample_str = aValue;}
private var _sample_str : String;


//PUBLIC CONST
/**
 * Comment for <code>PUBLIC_STATIC_CONSTANT</code>.
 *
 * @default PUBLIC_STATIC_CONSTANT
*/
public static const PUBLIC_STATIC_CONSTANT : String =
                    "PUBLIC_STATIC_CONSTANT";


//PRIVATE
/**
 * Comment for _sample2_str.
*/
private var _sample2_str : String;


//--------------------------------------
//  Constructor
//--------------------------------------
/**
 * This is the typical format of a simple multiline comment
 * such as for a <code>TemplateClass</code> constructor.
 *
 * <span class="hide">Any hidden comments go here.</span>
 *
 * @param param1 Describe param1 here.
 * @param param2 Describe param2 here.
*/
public function TemplateClass ()
{
      //SUPER
      super ();

      //VARIABLES
      var localSample_str : String =
            TemplateClass.PUBLIC_STATIC_CONSTANT;

      //PROPERTIES
      _sample_str = TemplateClass.PUBLIC_STATIC_CONSTANT;
      _sample2_str = TemplateClass.PUBLIC_STATIC_CONSTANT;

      //METHODS
      sampleMethod (_sample_str, _sample_str);

      //EVENTS
      addEventListener (Event.INIT, onInit);



}


//--------------------------------------
//  Methods
```

```as3
//-------------------------------------
//PUBLIC
/**
 * This is the typical format of a simple multiline comment
 * such as for a <code>sampleMethod</code> method.
 *
 * <span class="hide">Any hidden comments go here.</span>
 *
 * @param param1 Describe param1 here.
 * @param param2 Describe param2 here.
 *
 * @return void
*/
public function sampleMethod (aArgument_str : String,
                              aArgument2_str : String) : void
{
      sampleMethod2 (aArgument_str, aArgument2_str);
}


//PRIVATE
/**
 * This is the typical format of a simple multiline comment
 * such as for a <code>sampleMethod</code> method.
 *
 * <span class="hide">Any hidden comments go here.</span>
 *
 * @param param1 Describe param1 here.
 * @param param2 Describe param2 here.
 *
 * @return void
*/
private function sampleMethod2 (aArgument_str : String,
                                aArgument2_str : String) : void
{
      dispatchSample ();
}


//-------------------------------------
//  Events
//-------------------------------------
//Event Dispatchers
/**
 * Dispatches the Event.SAMPLE event.
*/
[Event(name="SAMPLE", type="flash.events.Event")]
private function dispatchSample () : void
{
      //dispatchEvent ( new Event (Event.SAMPLE) );

}


//Event Handlers
/**
 * Handles the Event.INIT event.
*/
private function onInit (aEvent : Event) : void
{
```

```
                }

        }
}
```

3.2. Style

3.2.1. Line and Line Wrap

When an expression doesn't fit on only one line (as defined in section 5, Style), break it on more than one line. In these cases, the line break must follow these rules:

- Break it after a comma
- Break it before an operator
- Prefer line break at higher level code
- Align the new line at the start of the previous line; if the previous rule isn't a good option, indent with two tabs.

Prefer:

```
// line #1: line break before the implements operator
// line #2: line break after a comma
// lines #2 and #3: indented with two tabs
public class Button extends UIComponent implements IDataRenderer,
                    IDropInListItemRenderer,
                    IFocusManagerComponent
{

        ...

}
```

Avoid:

```
public class Button extends UIComponent implements
                    IDataRenderer, IDropInListItemRenderer,
                    IFocusManagerComponent
{

        ...

}
```

Prefer:

```
// line #1 break at higher level, occurs outside the parentheses
// line #2 break doesn't break what is inside the parentheses
variable1 = variable2 + (variable3 * variable4  variable5)
                    - variable6 / variable7;
```

Avoid:

```
// line #1 break splits the parentheses contents in two lines
variable1 = variable2 + (variable3 * variable4
                    - variable5) - variable6 / variable7;
```

Line break example with ternary operators. Use only if too long to fit on one line:

```
value = (very_long_name_for_an_expression) ? value2
: value3; // aligned-left, ternary-unique!
```

3.2.2. Declarations

Do only one declaration per line:

Right:

```
var a_uint : int = 10;

var b_uint : int = 20;

var c_uint : int = 30;
```

Wrong:

```
var a_uint : int = 10, b_uint : int = 20, c_uint : int = 30;
```

Don't initialize public and private variables in its declaration (except for temporary variables and constants). Initialize instead in the constructor of the variable's containing class. Initialize the variable even if it is the default value.

Variables declarations should come on block beginning, except for variables used in loops.

```
public function getMetadata () : void
{

	var value_uint : uint = 123; // method-block beginning

}

public function getMetadata () : void
{

	if (_condition_boolean) {

		var value_uint : uint = 456; // if-block beginning

	}

}

public function getMetadata () : void
{

	//for-block beginning
	for (var i : uint = 0; i < valor; i++) {

		...

	}
}
```

Don't declare variables with names that were used before in another block, even if with different scope.

3.2.3. Curly Braces and Parentheses

Styling rules:

- **Do** put a space between the method name and the opening parentheses.
- **Do** put a space between the variable's name and its type.
- **Do** open curly braces on the line below class declarations.
- **Do** open curly braces on the line below method declarations.
- **Do** open curly braces on the line below event declarations.
- **Do** open curly braces on the same line in all other cases (if, for, while, do,

etc.).

- **Do** close curly braces in its own line at the same position in which the open curly brace is.
- **Do** separate methods by an empty line.
- **Don't** put a space between the parentheses and the method's arguments.

```
public class Example extends UIComponent implements IExample
{

        private var _item_object : Object;

        public function addItem (aX_uint : uint, aY_uint : uint) : void
        {

                if (true) {

                        ...

                } else {

                        ...

                }

                for (var i : uint = 0; I < = 10; i++ ) {

                        ...

                }

                while (true) {

                        ...

                }
        }

}
```

3.2.4. Statements

Simple

Simple statements must be one per line and should end with a semicolon.

Right:

```
i_uint++;

resetModel ();
```

Wrong:

```
i_uint ++; resetModel ();
```

Compound

Compound statements (the ones that typically use "{" and "}", like switch, if, while, etc.) must follow these rules:

- The code inside the statement must be indented by one level
- The curly brace must be on a new line after the declaration's beginning, aligned at the same position. The curly braces are closed in its own line, at the same position as the curly brace that opened the statement.
- Curly braces are used in all statements, even if it's only a single line.
- Always use the use "{" and "}" even when optional

Right:

```
if (isAvailable_boolean) {

        ...

}
```

Wrong:

```
if (isAvailable_boolean)  ...
```

Return

The return doesn't need to use parentheses unless it raises the understandability:

```
return;

return getFinalImage ();

return (phase ? phase : initPhase);
```

The return should occur only once in a method.

Conditionals with Simple Expressions

```
if (condition){

        ...

}


if (condition) {

        ...

} else if (condition) {

        ...

} else {

        ...

}
```

Conditionals with Compound Expressions

When the conditional expression is made of multiple variables, use parentheses and spacing as needed for readability.

```
//treat multiple simple expressions as just simple expressions,
//not special formatting required
if (!condition1 && condition2){

        ...

}

//use parenthesis and spacing for compound expressions
if (!condition1 && (expression1 == expression 2) ){

        ...

}
```

Conditional switch, case

Switch statements have the follow style:

```
switch (condition) {
        case 1:
                ...
                break;
        case 2:
                ...
                break;
        default:
                ...
                break;
}
```

Break rules:

- Always use break in the default case. Normally it's redundant, but it reinforces the idea.
- If a case doesn't have a break, add a comment where the break should be.

Loop for

```
for (initialization; condition; update) {
        ...
}
```

Loop for...in

```
for (var iterator : Type in someObject) {
        ...
}
```

Loop for each...in

```
for each (var iterator : Type in someObject) {
        ...
}
```

Loop while

```
while (condition) {
	...
}
```

Loop do...while

```
do {
	...
} while (condition);
```

Error handling try...catch...finally

```
try {
	...
} catch (aError : Error) {
	...
}
```

It can have the finally statement:

```
try {
	...
} catch (aError : Error) {
	...
} finally {
	...
}
```

3.2.5. Spaces

Wrapping lines

Line breaks make the code clearer, creating logical groups.

Use a line break:

- Between a code's logical section to make it clearer
- Between functions
- Between the method local variables and its first statement
- Before a block
- Before a single-line comment or before a multi-line comment about a specific code passage

Blank spaces

Use blank spaces to separate a keyword from its parentheses and **don't** use a space to separate the method name from its parentheses.

```
while (true) {
        getSomething ();
}
```

A blank space must exist after every comma in an arguments list:

```
addSomething (data1, data2, data3);
```

All binary operators (the ones with two operands: +, - , =, ==, etc.) must be separated from their operands by a space. **Don't** use a space to separate unary operators (++, -- , etc.).

```
a += (5 + b) / c;
while (d < MAX_COUNT) {
        i++;
}
```

Ternary operators must be separated by blank spaces and broken on more than one line if necessary:

```
a = (expression) ? expression : expression;
```

The for *expressions must be separated by blank spaces:*

```
for (expr1; expr2; expr3) {
    ...
}
```

3.3. Comments

3.3.1. Documentation Comments

Documentation comments are for classes, interfaces, variables, methods, and metadata with one comment per element before its declaration.
The documentation comment is meant to be read –and fully comprehended – by someone who will use the component but doesn't necessarily have access to the source code.

The comment format is the same read by ASDoc, and the syntax defined in the document: http://labs.adobe.com/wiki/index.php/ASDoc:Creating_ASDoc_Comments.

Example:

```
/**
* This is the typical format of a simple multiline comment
* such as for a <code>sampleMethod</code> method.
*
* <span class="hide">Any hidden comments go here.</span>
*
* @param param1 Describe param1 here.
* @param param2 Describe param2 here.
*
* @return void
*/
public function sampleMethod (aArgument_str : String,
                              aArgument2_str : String) : void
{

        sampleMethod2 (aArgument_str, aArgument2_str);

}
```

3.3.2. Implementation Comment

An implementation comment has the intention to document specific code sections that are not evident. The comments must use the // format, whether they are single or multiline.

If it's going to use a whole line, it must succeed a blank line and precede the related code:

```
// bail if we have no columns
if (!visibleColumns_array || (visibleColumns_array.length == 0) ) {

        ...

}
```

The comment can be in the same code line if doesn't exceed the line's maximum size:

```
columns_num = 0; // visible columns compensate for offset
```

NEVER use a comment to redundantly state what is obvious in the code:

```
columns_num = 0; // sets column numbers variables to zero
```

4. MXML

4.1. File Organization

MXML must contain the following structure:

#	Element	Obs.
1	XML Header: <?xml version="1.0" encoding="UTF8"?>	Always declare the encoding in the XML header, and always use UTF8.
2	Root component	Must contain all namespaces used in the file.
3	Metadata 1. Event 2. Style 3. Effect 4. Other metadata in alphabetical order	
4	Style definitions	Prefer external style files.
5	Scripts	Use only one Script block.
6	Non visual components	
7	Visual components	

4.2. Style

4.2.1. Line and Line Wrap

Use blank lines to make code clearer by visually grouping components. Always add a blank line between two components that are children of the same parent component if at least one of them (including their children) uses more than one line.

```
<mx:series>
	<mx:ColumnSeries yField="prev" displayName="Forecast">
		<mx:stroke>
			<mx:Stroke color="0xB35A00" />
		<mx:stroke>

		<mx:fill>
			<mx:LinearGradient angle="0">
				<mx:entries>
					<mx:GradientEntry ... />
					<mx:GradientEntry ... />
				</mx:entries>
			</mx:LinearGradient>
		</mx:fill>
	</mx:ColumnSeries>

	<comp:ColumnSeriesComponent />

</mx:series>
```

I.e., if a component has only one child, there's no need to insert a blank line. The below LinearGradient contains only one child entry.

```
<mx:LinearGradient angle="0">
	<mx:entries>
		<mx:GradientEntry ... />
		<mx:GradientEntry ... />
	</mx:entries>
</mx:LinearGradient>
```

Equally, as the entries children fit on one line, there's no blank line between them.

```
<mx:entries>
	<mx:GradientEntry ... />
	<mx:GradientEntry ... />
</mx:entries>
```

4.2.2. Nesting Components

Children components must be indented by their parent component:

```
<mx:TabNavigator>
	<mx:Container>
		<mx:Button />
	</mx:Container>
</mx:TabNavigator>
```

4.2.3. Attributes

Order the attributes by:

- Property
  - **Do**: The first property must be the id, if it exists;
  - Remember that width, height, and styleName are properties, not styles.
- Events
- Effects
- Style

If exist, the id attributes must be the first declared:

```
<mx:ViewStack id="mainModules">
```

To add readability, it is preferred to place one attribute per line, unless NONE of the attributes are likely to be edited. In that case, one line is preferred. For a likely to-be-edited attribute-set, use 1 line per attribute, and line-wrap the closing tag

```
<mx:Label
id = "custom_label"
	width = "100%"
	height = "100%"
	text = "Here comes a long enough text that... "
/>
```

For an unlikely to-be-edited attribute-set, use one line, but only if the entirety fits

on one line.

Few attributes:

```
<mx:Label id = “custom_label” width = “100%” />
```

The attributes must be indented by the component’s declaration if it uses more than one line.

```
<mx:Label
id = “custom_label”
        width = “100%”
/>
```

4.2.4. Script

In general coding, conventions dictate that coding within MXML is generally to be avoided in favor of code-behind techniques. For exceptions, though, this is the style for the Script block of ActionScript 3.0 in the *.mxml file:

```
<mx:Script>
        <![CDATA[
                ...
        ]]>
</mx:Script>
```

4.3. Comments

4.3.1. Documentation Comments

ASDoc tool doesn't support documentation comments in MXML files. But doing it is encouraged if the MXML file is a component that could be reused (and not only a simple view). This way, the file should contain an ActionScript comment inside a Script block.

```
<mx:Script>
	<![CDATA[
		/**
		•	Documentation comment inside a MXML component
		•	Uses the same format as the AS comment
		*/
	]]>
</mx:Script>
```

4.3.2. Implementation Comments

Use implementation comments to describe interface elements where their purpose or behavior is unclear.

```
<!— only shows up if is in admin role >
```

Or multiline comments:

```
<!Multiple
	line comments...
	...
	...
>
```

5. STYLE

5.1. General Rules

- Indent using tabs. The tab reference size is 4 spaces, and it's suggested to configure the IDE this way.
- Code lines must not exceed *100* characters [1].

[1] Using a 1280 pixels-wide resolution (ideal for 17" displays) with Eclipse, if 70% width is available to code (and the other 30% to Navigator), the line has about 103 character positions. The limit to print an A4 page is 80 characters.

6. NAMING

6.1. General Rules

- In general use the 'long' form of naming things. Code-completion (such as intellisense) in code editors save you from having to retype the full variable names in most cases. Time spent manually typing these names is negligible compared to time lost from confusion of less consistent naming schemes where sometimes abbreviations are used and sometimes they are not used. See acronyms* for exceptions to this rule.
- *Acronyms: avoid acronyms unless the abbreviation form is more usual than its full form (like URL, HTML, etc). Project names can be acronyms if this is the way it's called.
- **Do** use only ASCII characters and underscores.
- **Don't** use accents, spaces, punctuation marks, or special characters.
- **Don't** use the name of a native Flex SDK or Flash Player class name for a custom class name. It is perfectly fine (actually, it's preferred) to use similar names for objects (ex. `var bitmap : Bitmap = new Bitmap();`).
- **Don't** use 'Index' within a component name since it conflicts with ASDoc tool generated docs.

6.2. Language

The assumed audiences of this document are teams whose primary spoken language is English. Thusly, the code itself must be in English, except for verbs and nouns that are part of the *business domain* (specific expertise area the software is meant to address, i.e., the real-world part that is relevant to the system). In general use English for all parts of all coding and documentation. This convention is meant to standardize the code.

Right:

```
DEFAULT_CATEGORIA

getProdutosByCategoria ();

changeState ();
```

Wrong:

```
CATEOGRIA_POR_DEFECTO

obtenerProductosDeCategoría ();

mudarEstado ();
```

6.3. Packages

The package name must be written in *lowerCamelCase,* starting with small caps and other initials in upper case.

The first element in a package name is the first level domain. Here is a partial list.

- com
- org
- mil
- edu
- net
- gov

The next element is the company or client that owns the package, followed by the project's name and module:

```
com.company.project.module
```

Examples:

```
import com.adobe.effects.FullScreenWipe; //a fictitious reference
```

6.4. Classes

Class names should preferably be nouns, but can use adjectives as well. Always use *UpperCamelCase*.

Examples:

```
class LinearGradient
class DataTipRenderer
```

6.5. Interfaces

Interface names must follow the class naming rules, with a starting uppercase “I”.

Examples:

```
interface ICollectionView
interface IStroke
```

6.6. Methods

Methods must start with a verb and are written in *lowerCamelCase.* If the method is called on an event, it should end with *Handler:*

Examples:

```
makeRowsAndColumns ();

getObjectsUnderPoint ();
```

6.7. Events

Events follow the same convention as methods; however, they are to begin with ‘on.’

Examples:

```
onMouseDown ();

onSignUpComplete ();
```

6.8. Variables

The variable naming conventions have been developed to help developers know at a glance the most information possible from within a class (variable type and variable scope), and properly hide details from outside the class. Overall variables must use *lowerCamelCase* and objectively describe what they are. Most, by default, end with a suffix describing the type of

variable for both native and custom types. Here is a breakdown of various variable types.

Private variables

Private variables must start with an underscore "_". This is to help distinguish them, which aids the developer in being conscious of the private variables vs. the public interface of classes.

```
private var _sample_str : String;
```

Native Variable Suffixes

```
_message_str
_index_int
_phoneNumber_uint
_floatingCoordinant_num
```

Here is a partial list of suffixes. In general, use the full lower-cased, unabbreviated class name as the extension. There are exceptions due to legacy Macromedia conventions.

Object type	Variable suffix
Array	_array
Button	_button
Boolean	_boolean
Camera	_camera
Color	_color
Date	_date
Error	_error
Integer	_int
LocalConnection	_localconnection
Microphone	_microphone
MovieClip	_mc
PrintJob	_pj
NetConnection	_netconnection
NetStream	_netstream
Number (Float)	_num

SharedObject	_sharedobject
Sound	_sound
String	_str
TextField	_textfield
TextArea	_textarea
Text	_text
Unsigned Integer	_uint
Video	_video
XML	_xml
XMLNode	_xmlnode
XMLSocket	_xmlsocket

Custom variable suffixes

The spirit of the conventions follows here -- same casing and same idea of an extension to indicate type. If there is more than one variable of the same type in the same scope, the type is used as the extension.

```
private var _topExterior_windowbox : WindowBox;
```

If there is NOT more than one variable of the same type in the same scope, the type can be used as the variable name WITHOUT an extension. This is not required, but is helpful to support the readability that there is indeed only one and only variable of its type.

```
private var _overallInstanceManager : OverallInstanceManager;
```

Public variables

Public variables must **not start** with an underscore "_" and must **not end** with an extension. Avoid using public variables in general, as it does not properly encapsulate the software design; instead use getter/setter conventions (see next section). The philosophy here is that 'inside' a class (private variables) the naming is more verbose to include an extension that denotes the variable type. However, from 'outside' a given class (public

members) should hide details of implementation include variable type so no extension is to be used. This supports the OOD methodology of encapsulation,

```
public var sample : String;
```

Getter/Setters

Typically getter/setters should occupy only three lines, if margins allow, in the order shown below. In more complex situations, more than three lines may be needed, but that is the exception. The convention of 'aValue' is not changed, not made more descriptive, nor given a suffix. This keeps it short and consistent across all getter/setters. Do **not** use a variable extension for the public interface. This properly hides the variable type from the API-user. Implementation details such as that should not be shared by default. [Note: This code sample is left-aligned to fit within margins, in this document. In your code, indent getter/setters as you would any other code.]

```
public function get name () : String { return _name_str; }
public function set name (aValue : String) : void { _name_str = aValue;}
private var _name_str : String;
```

Boolean Variables

Boolean variables should start with *can*, *is,* or *has*.

```
private var isListeningForRender_boolean : Boolean = false;

private var canEditUsers_boolean : Boolean = true;

private var hasAdminPrivileges_boolean : Boolean = false;
```

Temporary Variables

Temporary variables follow the same conventions as privates, but without the prefix "_".

```
var sample_str : String;
```

Temporary Variables in Loops

Temporary variables (like the ones used in loop statements) should be only one character. The most commonly used are i, j, k, m, n, c, d.

```
for (var i : uint = 0; i < 10; i++) {

	...

}
```

Arguments

Arguments, which are variables passed into a function, method, or event, accessed from within that scope, begin with ‘a’

```
//constructor
public function Book (aTotalChapters_uint : Number = 0) : void {

	...

}


//method
public function viewPageByNumber (aPage_uint : Number = 1) : void {

	...

}

//event
public function onOpenNewPage (aCurrentPage_uint : uint) : void {

	...

}
```

6.9. Constants

Constants must be all upper case, splitting the words with an underscore (_). Constants are not given a suffix. Where applicable, a constant’s value should match the constant’s name. Constants are exceptional among variables in that they should be initialized with a value in the declaration.

```
public const DEFAULT_MEASURED_WIDTH : Number = 160;

public static const DEFAULT_MEASURED_WIDTH : Number = 160;

private static const AUTO : String = “AUTO”;

private const AUTO : String = “AUTO”;
```

6.10. Namespaces

Namespaces names must be all lower case, splitting the word with an underscore (_):

```
mx_internal

object_proxy
```

The file must have the same name as the namespace it defines.

7. GENERAL PRACTICES

7.1 Fixme and Todo

Use the keyword FIXME inside comments (MXML and ActionScript) to flag something that's broken and should be fixed. Use TODO to flag something that works but can be improved by a refactoring. For this, use the *Flex Builder Todo Plugin* available from a Google search.

Assign the iterator value to a variable before using it if the performance improvement will be significant (e.g., in simple arrays, it isn't necessary).

Right:

```
var maxPhase_uint : uint = reallySlowMethod ();

for (var i : Number = -10; i < maxPhase_uint; i ++) {

        ...

}
```

Right:

```
var months_array : Array = ['Jan','Fev','Mar'];

// it's quicker to calculate an array size outside a for
// but we're targeting readability too
for (var i : Number = 0; i < months_array.length; i++) {

        ...

}
```

Wrong:

```
for (var i : Number = 0; i < reallySlowMethod(); i++) {

        ...

}
```

It's encouraged in the creation and use of loose coupled components. The less a component knows about another, the greater the reuse possibilities.

In Boolean evaluations, place the fastest ones first.

7.2 Casting

ActionScript 3 supports not just compile-type type checking, but runtime type checking as well. If a conflict exists with types during runtime in ActionScript 3, a runtime error will be thrown. In general, this is a good thing. But it can also require that you sometimes massage your variable typing to prevent unwarranted errors. This is called 'casting' one variable of a given type to behave like it is of another type. There are two ways to do this. The 'as' operator, new to ActionScript 3.0, is the preferred technique.

Avoid:

```
var genericInstance_object : Object = new Object ();

methodThatExpectsAString ( String (genericInstance_object));
```

Prefer:

```
var genericInstance_object : Object = new Object ();

//cast as string to meet method's signature
methodThatExpectsAString ( genericInstance_object as String );

//to call a String method
(genericInstance_object as String).length; //returns the length
```

7.3 Typing to Interfaces

Typing variables allows the compiler to know what members (variables, methods, events) are available from its API.

- You have to list all public methods that will be common for the classes that implement this interface.
- You do so by writing the method's signature, definition, name, parameters, and return type.
- You don't have to open and close curly braces after the method definition.
- You don't have to (and can't) specify if the method is private, public, or protected, because an interface is only for public scope.
- interface can only extend other interfaces.
- A class can implement more than one interface.

Commonly, a programmer may set the type of a variable to the variable's class.

Example:

```
var bigMonsterSprite : BigMonsterSprite = new BigMonsterSprite ();
```

This works fine in many cases, but using interfaces allows many benefits. An interface is a 'promise' from a class to the compiler about exactly which of its methods are available from the API. When two or more classes use the same super class, it can be useful to treat them the same (polymorphism) from the API. However, if two classes do not use the same super class and thus cannot be typed using a class, an interface is useful. See the example below.

Example:

```
//in IMonster.as interface file
package
{

	public interface IMonster {

		function getShot(damage_uint : uint) : void;

		function getCanShoot() : Boolean;

	}

}


//in BigMonsterSprite.as class file
public class BigMonsterSprite extends MovieClip implements IMonster
{

	...

}


//in SmallMonsterSprite.as class file
public class SmallMonsterSprite extends MovieClip implements IMonster
{

	...

}
```

```
//in MonsterGame.as class file
var bigMonsterSprite : IMonster = new BigMonsterSprite ();
var smallMonsterSprite : IMonster = new SmallMonsterSprite ();


//variables typed as interfaces ‘promise’ a certain API so
//the compiler agrees these statements are valid
bigMonsterSprite.getShot (10);
smallMonsterSprite.getShot (20);
```

8. APPENDIX

8.1 Reserved Words

- ActionScript 3 reserved words are those words with special meanings in the language. **Don't** use them for your own custom naming to avoid conflicts.
- There are also syntactic keywords that have special meanings in some contexts, so these keywords should be avoided on produced code.
- There are future reserved words that should be avoided, too.

8.2 Challenges of Conforming to Standards

Seeing the value in standards and being vigilant to adhere to standards are two different things. Undisciplined teams, time pressures, and exhaustion can keep projects from meeting goals of code standardization. Beware of 'Broken Windows' – a sociological concept first proposed by James Q. Wilson and George L. Kelling. If code is not meeting standards, the tendency is for developers to 'break' conventions more and more, resulting in the slippery slope to a non-standards-compliant project.

Here are a few tips:

- **Start Fresh** – It is possible, but surely a challenge, to convert an existing project to conform to new standards. Consider waiting to start a new project to establish new conventions.
- **Use coding-file templates** – create generic or project-specific templates for common file types and use them to start each new file. Once you have a standards doc (like this one), using templates is the most valuable step to establish standards.
- **Budget time for standards-compliant code** – No amount of wishful thinking will help your team meet standards if you don't schedule enough time. Schedule 5-15% more time during development depending on how your team adapts.
- **Schedule code reviews** – Undoubtedly on any team, code-compliance discipline will vary and even the most dedicated developer will make mistakes. In addition to scheduling code reviews with the goal of bug-free, well-optimized code, build in checks for standards compliance.
    - **Encourage** developers to commit only compliant code.
    - **Use regular expressions** and other programmatic search techniques to find violations in compliance.
        - Search for "[^]\:[^]", for example, to find any violations of the convention to keep spaces around colons for instance (" : ").
    - **Setup a checklist** for code reviewers to follow. It may be a boiled-down version of a more complete standards doc, for a quick reference.

9. DOCUMENT HISTORY

- Release 1.0
  Samuel Asher Rivello, March 2008

  This is the first release with MXML and Actionscript guidelines. Focus on standardization and readability. This document is based on the PDF *Adobe Flex Coding Standards* by Fabio Terracini. It is also informed by Java programming guidelines, internal guidelines, the Flex 3 SDK's guidelines, and personal experience and preference.

- Release 1.1
  Samuel Asher Rivello, July 2008

  The doc has been modified based on feedback on readability, accuracy, and robustness.