# Adobe Flex Coding Guidelines

## MXML e ActionScript 3


*Adobe Flex programming coding style & guidelines at DClick*

*v1.2 – February/2007*
*Fabio Terracini*
*fabio.terracini@dclick.com.br*

# 1. Introduction

This document aims to establish coding guidelines to application written with Adobe Flex 2 and ActionScript 3.

Establish a coding convention matters in the context that the most part of the time in software development life cycle is maintenance. This way, helping the comprehension of code passages is a must, considering that not always who's going to perform the maintenance is the same person who built it in the first time. With a common language, developers can rapidly understand other person's code. Besides, the application or components code can be distributed or sold to third party.

The premises of coding conventions are:

- Consistency;
- Code comprehension.

The practices established in this document are based on DClick work way, Java coding conventions and conventions seen at Adobe Flex 2 SDK.

# 2. Files

## *2.1. Files suffixes*

- MXML code: .mxml
- ActionScript code: .as
- CSS code: .css

## *2.2. File names*

- Must not contain spaces, punctuations 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 UTF-8 format.

# 3. ActionScript 3.0

## *3.1. File organization*

An ActionScript file must contain the follow structure:

| # | **Element** | **Obs.** |
|---|---|---|
| 1 | Initial Comment | |
| 2 | Package definition | |
| 3 | Namespace declaration<br>• If has one, it is the last section | A file that defines a namespace only does that. |
| 4 | *Import* statements<br>1. Package *flash*<br>2. Package *mx*<br>3. Package *com.adobe*<br>4. Package *br.com.dclick* (DClick components)<br>5. Packages of third party in alphabetical order<br>6. Package of the project this files belongs<br><br>Use fully qualified imports, i.e., without the asterisk. Unless when a big part of the package are used.<br>• Prefer: `import mx.core.Application;`<br>• Avoid: `import mx.core.*;` | Inside these sections, all imports must in alphabetical order.<br><br>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<br>1. Event<br>2. Style<br>3. Effect<br>4. Other metadata in alphabetical order. | |
| 7 | Class or interface definition | |
| 8 | *static* variables<br>1. *public*<br>a. *const*<br>b. Others *public static*<br>2. *internal*<br>3. *protected*<br>4. *private*<br>5. custom namespaces<br>a. In alphabetical order | |
| 9 | Instance variables aren't handled by getters and setters<br>1. *public*<br>2. *internal* | |

| | | |
|---|---|---|
| | 3. *protected*<br>4. *private*<br>6. custom namespaces<br>a. In alphabetical order | |
| 10 | Constructor | |
| 11 | Getters and setter managed variables and the *get* and *set* methods themselves, as related variables. Example:<br><br>`private var _enabled:Boolean = true;`<br>`private var enabledChanged:Boolean = false;`<br><br>`public function get enabled():Boolean`<br>`{`<br>`  return _enabled;`<br>`}`<br><br>`public function set enabled(value:Boolean):void`<br>`{`<br>`  _enabled = value;`<br>`  enabledChanged = true;`<br>`}` | See the variables section on this document for rules about variables managed by *get* and *set* methods. |
| 12 | Methods | Grouped by functionality, not by scope. |

## *3.2. Style*

### 3.2.1. Line and line wrap

When an expression doesn't fit in only one line, break it in 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 break at higher level, occurs outside the parentheses
// line break doesn't break what is inside the parentheses
variable1 = variable2 + (variable3 * variable4 - variable5)
            - variable6 / variable7;
```

Avoid:

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

Line break example with ternary operators:

```
b = (expression) ? expression
                 : gamma;      // aligned!

c = (expression)
      ? beta
      : gamma;
```

### 3.2.2. Declarations

Do only one declaration per line:

Right:

```
var a:int = 10;
var b:int = 20;
var c:int = 30;
```

Wrong:

```
var a:int = 10, b:int = 20, c:int = 30;
```

Initialize the variable if possible. There's no need to initialize the variable if its start value depends on some process to occur. Initialize the variable even if it is the default value.

Right:

```
public var isAdmin:Boolean = false;
```

Wrong:

```
public var isAdmin:Boolean; // false is default for Boolean
```

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

```
public function getMetadata():void
{
	var value:int = 123;	// method block beginning

	...

	if (condition)
	{
		var value:int = 456;	// if block beginning
		...
	}

	for (var i:int = 0; i < valor; i++) // in the for
	{
		...
	}

}
```

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:

- Don't put a space between the method name and the opening parentheses, and don't put a space between the parentheses and the method's arguments;
- Don't put a space between the object name and his type;
- Open curly braces in a new line at the same position in which the method declaration begins;
- Close curly braces in its own line at the same position in which the open curly brace is.
- Methods are separated by an empty line.

```
public class Example extends UIComponent implements IExample
{

	private var _item:Object;
```

```
        public function addItem(item:Object):void
        {
              _item = item;
              ...
        }

        public function anotherMethod():void
        {
              while (true)
              {
                    ...
              }
        }

}
```

### 3.2.4. Statements

## Simple

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

Right:

```
i++;
resetModel();
```

Wrong:

```
i++; resetModel();
```

## Compound

Compound statements (the ones that require `{` 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 in a new line after the declaration's beginning, aligned at the same position. The curly braces are closed in its own line, at the position the curly brace that opened the statement.
- Curly braces are used in all statements, even if it's only a single line.

## Return

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

```
return;

return getFinalImage();

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

## Conditional if, else if, else

```
if (condition)
{
      simpleStatement;
}


if (condition)
{
      statements;
}
else
{
      statements;
}

if (condition)
{
      statements;
}
else if (condition)
{
      statements;
}
else
{
      statements;
}
```

## Conditional switch, case

Switch statements have the follow style:

```
switch (condition)
{

      case ABC:
      {
            statements;
            // continue, without break
      }
```

```
        case DEF:
        {
              statements;
              break;
        }

        case JKL:
        case XYZ:
        {
              statements;
              break;
        }

        default:
        {
              statements;
              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.
- It's not necessary to use a `break` if the statement contains a `return`.

## Loop for

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


for (initialization; condition; update);
```

## Loop for..in

```
for (var iterator:type in someObject)
{
      statements;
}
```

## Loop for each..in

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

```
        statements;
}
```

## Loop while

```
while (condition)
{
        statements;
}
```

## Loop do..while

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

## Error handling try..catch..finally

```
try
{
        statements;
}
catch (e:Type)
{
        statements;
}
```

It can have the finally statement:

```
try
{
        statements;
}
catch (e:Type)
{
        statements;
}
finally
{
        statements;
}
```

## With

```
with (this)
{
        alpha = 0.5;
```

```
}
```

### 3.2.5. Spaces

## Wrapping lines

Line breaks makes the code clearer, creating logical groups.

Use a line break when:

- 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;
- Between a code's logical section to make it clearer.

## 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 separate from its operands by a space. Don't use a space to separate unary operators (++, --, etc).

```
a += (5 + b) / c;

while (d as int < f)
{
       i++;
}
```

Ternary operators must be separated by blank spaces and broken in 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 that 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:

```
/**
 *  The Button control is a commonly used rectangular button.
 *  Button controls look like they can be pressed.
 *
 *  @mxml
 *
 *  <p>The <code>&lt;mx:Button&gt;</code> tag inherits all the
 *  tag attributes of its superclass, and adds the following:</p>
 *
 *  <pre>
 *  &lt;mx:Button
 *    <b>Properties</b>
 *    autoRepeat="false|true"
 *    ...
 *    <b>Styles</b>
 *    borderColor="0xAAB3B3"
 *    ...
 *  /&gt;
 *  </pre>
 *
 *  @includeExample examples/ButtonExample.mxml
 */
public class Button extends UIComponent
```

Another example:

```
/**
 *  @private
 *  Displays one of the eight possible skins,
 *  creating it if it doesn't already exist.
 */
mx_internal function viewSkin():void
{
```

### 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 || visibleColumns.length == 0)
```

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

```
colNum = 0; // visible columns compensate for firstCol offset
```

Never use comment to translate code:

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

# 4. MXML

## *4.1. File organization*

A MXML must contain the follow structure:

| # | **Element** | **Obs.** |
|---|---|---|
| 1 | XML Header:<br>`<?xml version="1.0" encoding="UTF-8"?>` | Always declare the encoding at the XML header, and always use UTF-8. |
| 2 | Root component | Must contain all namespaces used in the file. |
| 3 | Metadata<br>1. Event<br>2. Style<br>3. Effect<br>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 `entries`.

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

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

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

### 4.2.2. Nestling 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
  - The first property must be the `id`, if it exists;
  - Remember that `width`, `height` e `styleName` are property, not styles.
- Events

- Effects
- Style

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

```
<mx:ViewStack id="mainModules" height="75%" width="75%">
```

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

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

In declarations that use more than one line, the only attribute in the first line is the `id`. All other attributes must be in the next lines, ordered.

```
<mx:ViewStack id="mainModules"
      height="75%" width="75%"
      paddingTop="10" paddingLeft="10" paddingRight="10">

<mx:ViewStack
      height="75%" width="75%"
      paddingTop="10" paddingLeft="10" paddingRight="10">
```

Related attributes should be in the same line. In the follow example, the second line only contains properties, the third has events, fourth has only styles and the last one has only effects.

```
<mx:Panel
      title="VBox Container Example" status="Some status"
      hide="doSomething()" creationComplete="doSomething()"
      paddingTop="10" paddingLeft="10" paddingRight="10"
      resizeEffect="Resize" />
```

In cases where more than one line is needed for an attribute type, use more than one line (observing consistency, order and line size) keeping them grouped by type. In the following example, the first line contains the declaration plus id, second has only properties, third has events, fourth has some styles, fifth has padding styles (there are no effects).

```
<mx:Panel id="pnLoginInfo"
      title="VBox Container Example" height="75%" width="75%"
      resize="resizeHandler(event)"
      titleStyleName="titleLogin" headerHeight="25"
      paddingTop="10" paddingLeft="10" paddingRight="10" />
```

### 4.2.4. Script

This is the style for the `Script` block:

```
<mx:Script>
      <![CDATA[

            code;

      ]]>
</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 alike 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 it isn't clear what's its purpose or behavior.

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

Or multline 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

- 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;
- Only ASCII characters, no accents, spaces, punctuations or special characters;
- Don't use the name of a native Flex SDK component (from the mx package like Application, DataGrid, etc) neither from Flash Player (flash package, like IOError, Bitmap, etc);
- Since writing in MXML is a easy way to write in ActionScript, the naming rules in MXML are the same as in ActionScript (a MXML file is like an ActionScript class, and its internal components and variables are properties, for example);
- The main application file should be named Main.mxml
    - Never use Index to a component name since it conflicts with ASDoc tool generated docs.

## 6.2. Language

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).

This way, the follow examples are acceptable:

```
DEFAULT_CATEGORIA
addNotaFiscal()
getProdutosByCategoria()
changeState()
UsuarioVO
screens/Reports.mxml
```

The follow examples, although, aren't valid uses:

```
popularCombo()
mudarEstado()
UsuarioObjetoDeTransferencia
```

## 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 (com, org, mil, edu, net, gov) or a two letter code identifying a country as documented in ISO 3166 followed by the first level domain (br.com, ar.edu, uk.gov, etc).

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

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

Examples:

```
br.com.dclick.mediaManager.uploadModule

com.apple.quicktime.v2
```

## 6.4. Classes

Classes names should prefer nouns, but can use adjectives as well. Always use *UpperCamelCase*.

Examples:

```
class LinearGradient

class DataTipRenderer
```

## 6.5. Interfaces

Interface names must follow the classes 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:*

Exemples:

```
makeRowsAndColumns()

getObjectsUnderPoint()

mouseDownHandler()
```

## 6.7. Variables

Variables must use *lowerCamelCase* and objectively describe what they are. Variables must start with and underscore (_) if they are going to be manipulated by *get* and *set* methods.

```
private var _enabled:Boolean = true;

public function get enabled():Boolean
{
      return _enabled;
}

public function set enabled(value:Boolean):void
{
      _enabled = value;
}
```

There are no variable prefixes. ActionScript has typed object and a clear and concise name is more important than the object's type. Although, Boolean variables should start with *can*, *is* or *has*.

```
private var isListeningForRender:Boolean = false;

private var canEditUsers:Boolean = true;

private var hasAdminPrivileges:Boolean = false;
```

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`. The lowercase "L" must not be used.

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

The `catch` variables must be `e`, no matter what's the error type (if it has a class describing it).

```
catch (e:Error)
{
      hasFieldName = false;
      ...
}
```

## 6.8. Constants

Constants must be all uppercase, splitting the words with an underscore (_):

```
public static const DEFAULT_MEASURED_WIDTH:Number = 160;

public static const AUTO:String = "auto";
```

## 6.9. Namespaces

Namespaces names must be all lowercase, 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

- 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 2 Task Plug-in*.

- 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:int = reallySlowMethod();

for (var i:Number = 0; i < maxPhase; i++)
{
      statements;
}
```

  Right:

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

// it's quicker to calculate an array size
// we're targeting readability too
for (var i:Number = 0; i < months.length; i++)
{
      trace(months [i]);
}
```

  Wrong:

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

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

- In Boolean evaluations, place the fastest ones before.

  Right:

```
if (isAdmin && slowMethod(item))
```

  Wrong:

```
if (slowMethod(item) && isAdmin)
```

- Use constants if available.

  Right:

```
myButton.addEventListener(MouseEvent.CLICK, myHandler);
```

  Wrong:

```
myButton.addEventListener("click", myHandler);
```

- Use specific types if available.

  Right:

```
private function mouseMoveHandler(event:MouseEvent):void
```

  Wrong:

```
private function mouseMoveHandler(event:Event):void
```

- Prefer anonymous functions for simple event handler, with only one statement. The follow styles are permitted.

```
myBytton.addEventListener(MouseEvent.CLICK,
      function(event:MouseEvent):void
      {
            simpleStatement;
      }
);

myBytton.addEventListener
(
      MouseEvent.CLICK,
      function(event:MouseEvent):void
      {
            simpleStatement;
      }
);
```

# 8. Appendix: Reserved Words

The follow table contains ActionScript 3 reserved words:

| | | | |
|---|---|---|---|
| as | break | case | catch |
| class | const | continue | default |
| delete | do | else | extends |
| false | finally | for | function |
| if | implements | import | in |
| instanceof | interface | internal | is |
| native | new | null | package |
| private | protected | public | return |
| super | switch | this | throw |
| to | true | try | typeof |
| use | var | void | while |
| with | | | |

There are also syntactic keywords that have special means in some contexts so these keywords should be avoided on produced code:

| | | | |
|---|---|---|---|
| each | get | set | namespace |
| include | dynamic | final | native |
| override | static | | |

And there are future reserved words that should be avoided too:

| | | | |
|---|---|---|---|
| abstract | boolean | byte | cast |
| char | debugger | double | enum |
| export | float | goto | intrinsic |
| long | prototype | short | synchronized |
| throws | to | transient | type |
| virtual | volatile | | |

# 9. Document history

**Release 1.2**
*Fabio Terracini, February 9, 2007*

English translation and some typos fixed.

**Release 1.1**
*Fabio Terracini, December 13, 2006*

Detailed line wrap (item 3.2.1). Get and set examples on variables. Compound statements example. Puntuactions fixes.

**Release 1**
*Fabio Terracini, December 6,/2006*

First release with MXML and Actionscript guidelines. Focus on standardization and readability. Based on Java guidelines, DClick internal guidelines and also on Flex 2 SDK.