

ActionScript 2 Coding Standards

The Method

by

Simon Wacker

(<http://www.simonwacker.com/blog/archives/000087.php>)

15. April 2005

Method Documentation

Example:

```
/**
 * Returns an array that contains the methods represented by {@link MethodInfo}
 * instances, this type and super types' declare, that are not filtered/excluded.
 *
 * <p>The {@link TypeMemberFilter#filter} method is invoked for every
 * method to determine whether it shall be contained in the result. The
 * passed-in argument is of type {@code MethodInfo}.
 *
 * <p>If the passed-in method filter is {@code null} or {@code undefined}
 * the result of an invocation of the {@link #getMethodsByFlag} method
 * with argument {@code false} will be returned.
 *
 * <p>{@code null} will be returned if:
 * <ul>
 * <li>The {@link #getType} method returns {@code null} or {@code undefined}.</li>
 * <li>The {@link #getMethodsByFlag} method returns {@code null} or {@code undefined}.
 * </li>
 * </ul>
 *
 * @param methodFilter the filter that filters unwanted methods out
 * @return an array containing the declared methods that are not filtered,
 * an empty array if no methods are declared or all were filtered or null
 * @see #getMethodsByFlag
 */
public function getMethodsByFilter(methodFilter:TypeMemberFilter):Array {
    if (!getType()) return null;
    if (!methodFilter) return getMethodsByFlag(false);
    var result:Array = getMethodsByFlag(methodFilter.filterSuperTypes());
    for (var i:Number = 0; i < result.length; i++) {
        if (methodFilter.filter(result[i])) {
            result.splice(i, 1);
            i--;
        }
    }
    return result;
}
```

Constitution: The method documentation begins with the begin-comment delimiter „/**“ and ends with the end-comment delimiter „*/“. Both comment delimiters are placed on their own line. Every line in-between has a leading asterisk character „*“ which is indented with a blank space to line it up. The line length of the documentation text should be limited to 80 characters, regardless of the preceding indentation and asterisk character. The method documentation is composed of a main description followed by a tag section. Both these parts are allowed to contain in-line tags.

Good Example:

```
/**
 * [Main Description]
 *
 * [Tag Section]
 */
```

Bad Example:

```
/** [Main Description]
 * [Tag Section]
 */
```

or:

```
/**
 *
 * [Main Description]
 *
 * [Tag Section]
 */
```

Position and Alignment: The method documentation precedes the method declaration and is indented to align with it. There is no blank line between the documentation and the declaration.

Good Example:

```
/**
 * ...
 */
public function getName(Void):String;
Bad Example:
/**
 * ...
 */

public function getName(Void):String;
```

Speech: Write the description in 3rd person declarative (descriptive) rather than 2nd person imperative (prescriptive).

```
Good Example:
Returns an array that ...
Bad Example:
Return an array that ...
```

When you refer to the instance created from the current class use „this“ instead of „the“.

```
Good Example:
Returns the log level of this logger.
Bad Example:
Returns the log level of the logger.
```

Avoid Latin. This means use „for example“ instead of „e.g.“, „also known as“ instead of „aka“ and so on.

Main Description: The main description starts in the line after the begin-comment delimiter. The first sentence of it is a short summary, starting with a verb, that contains a concise but complete description of the method. The summary is followed by a blank line and paragraphs, separated by „<p>“ tags that provide more information about the method. There is a blank line before every „<p>“ tag. Document special cases like what happens if a specific parameter is „null“ or if a dependent method returns „null“. Try to be as concise as possible so that developers are able to use the method extensively by only reading its documentation. You should nevertheless try to write the description as implementation-independent as possible and specify dependencies only where necessary, because the documentation defines a contract the method and any over-writing or implementing methods must meet. That means the more implementation-specific the contract the harder it is to refactor the method without violating it.

```
Good Example:
/**
 * Short summary sentence that starts with a verb.
 *
 * <p>Paragraph one that is separated by a {@code <p>} tag. It may span
 * multiple lines and contain multiple sentences. Limit the line length
 * to 80 characters.
 *
 * <p>Paragraph two that is also separated by a {@code <p>} tag.
 *
 * <p>{@code null} will be returned if:
 * <ul>
 * <li>Case one.</li>
 * <li>Case two.</li>
 * <li>Case three.</li>
 * </ul>
 *
 * [Tag Section]
 */
```

```
Bad Example:
/**
 * Long description of what the method does. It is not concise but just says about every-
 * thing in an awkward manner. The line length exceeds 80 characters.
 * <p>There is no blank line before the <p> tag. This makes reading the documentation
 * more difficult. In-line code is not enclosed in {@code} tags. Special cases are not
 * mentioned.
 * [Tag Section]
 */
```

Tag Section: The tag section is separated from the main description by a blank line. It starts with the first occurrence of an „@“ character that institutes a block tag. This section provides information about parameters „@param“, return values „@return“ and exceptions „@throws“ of the method. It also contains „@see“ tags to point to references, like dependent methods.

Good Example:

```
/**
 * [Main Description]
 *
 * @param parameterOne the first parameter that ...
 * @param parameterTwo the second parameter that ...
 * @return the ...
 * @throws IllegalArgumentException if {@code parameterOne} is {@code null}
 * @see #doSomethingSimilar
 * @see org.as2lib.env.reflect.ClassInfo#forInstance
 */
public function doSomething(parameterOne:String, parameterTwo:Number):String;
```

Bad Example:

```
/**
 * [Main Description]
 *
 * @param parameterTwo the second parameter that ...
 * @return the ...
 * @see org.as2lib.env.reflect.ClassInfo#forInstance
 */
public function doSomething(parameterOne:String, parameterTwo:Number):String;
```

Block Tags Ordering: Include block tags in the following order:

```
@param parameter-name description
@return description
@throws class-name description
@see „string“
@see <a href="url#value">label</a>
@see package.class#member label
@deprecated deprecated-text
```

@param parameter-name description: „@param“ tags are ordered the same as the parameters in your method declaration. „parameter-name“ is the name of the parameter in your method declaration. „description“ is a partial sentence that describes the parameter. It starts with a lower case word and does not end with a period. It is allowed to span multiple lines. If the parameter does not have to be passed-in, that means is optional, put the „(optional)“ text after „parameter-name“ and before „description“.

Example:

```
@param [parameter-name] (optional) [description]
```

If the method expects an unknown number of parameters signal this in your documentation with an „@param“ tag and „..“ as „parameter-name“. Add a description as for any other parameter. If the parameters must be of a specific type mention that type in your description.

Example:

```
@param .. [description]
```

@return description: Include „@return“ tags for all methods that do not return „Void“ and that are not constructors. „description“ is a partial sentence that describes the return type and permissible range of values. It starts with a lower case word and does not end with a period. It is allowed to span multiple lines. Supply return values for special cases whenever possible. Such a special case is for example returning „null“.

@throws class-name description: „class-name“ is the name of the exception that may be thrown by the method. „description“ is a partial sentence that says in which cases the exception is thrown. It starts with a lower case word and does not end with a

period. It is allowed to span multiple lines. This tag is very important and must not be forgotten because if not specified there is no possibility of knowing what exceptions are thrown and in which cases besides searching in the whole method implementation for „throws“ clauses directly or getting an exception at run-time.

Example:

```
@throws UnknownOverloadHandlerException if no adequate overload handler was found
```

@see reference: Take a look at the „@see“ tag description on <http://java.sun.com/j2se/1.5.0/docs/tooldocs/windows/javadoc.html#@see> and at how to order „@see“ tags on <http://java.sun.com/j2se/javadoc/writingdoccomments/#multiple@see>.

In-Line Tags: As said before, both the main description and the tag section are allowed to contain in-line tags. In-line tags appear within curly braces as „{@tag}“. They can be used anywhere that text is allowed. Common in-line tags are „{@link}“ to point to a reference and „{@code}“ to highlight code.

Example:

```
The {@link TypeMemberFilter#filter} method is invoked ...
```

or:

```
If the passed-in method filter is {@code null} or ...
```

If you refer to or highlight a method omit the parentheses and use only the method name. This makes reading the resulting method documentation more natural and easy.

Good Example:

```
The {@link TypeMemberFilter#filter} method is invoked ...
```

Bad Example:

```
The {@link TypeMemberFilter#filter(MethodFilter):Boolean} method is invoked ...
```

{@link package.type#member label}: Inserts an in-line link with visible text „label“ that points to the documentation for the specified package, type or member of a referenced type. The label is optional. If you do not specify it the API name will be used. Do not add a link to all API names in your documentation. Because links call attention to themselves it can make the documentation more difficult to read if used profusely. Link the same API name in one method documentation only ones. You may also only highlight the API names using the „{@code}“ tag and provide „@see“ tags for them in the tag section.

Example:

```
The {@link #getMethodsByFlag} method returns ...
```

{@code text}: Displays text in code font. Use „{@code}“ tags to highlight API names. These include parameter names and constants like „null“ or „undefined“ as well as class, interface, package, property, variable and method names.

Example:

```
{@code null} will be returned if ...
```

<code>text</code>: Displays text in code font. Use this html tag if the code spans multiple lines and stands on its own, that means is not directly included in a sentence. The opening tag „<code>“ is followed by a return. The following code is indented with two spaces. Do also not use tabs in your code but replace them with four spaces. The closing tag „</code>“ is placed on its own line and is aligned with its corresponding opening tag. It ends the code.

Example:

```
<code>
    var server:LocalServer = new LocalServer("local.as2lib.org");
    server.putService("myServiceOne", new MyServiceOne());
    server.putService("myServiceTwo", new MyServiceTwo());
    server.run();
</code>
```

or:

```
<code>
```

```

var client:LocalClientServiceProxy = new
LocalClientServiceProxy("local.as2lib.org/myService");
var callback:MethodInvocationCallback = client.invoke("myMethod", ["p1", "p2"]);
callback.onReturn = function(returnInfo:MethodInvocationReturnInfo):Void {
    trace("myMethod - return value: " + returnInfo.getReturnValue());
}
callback.onError = function(errorInfo:MethodInvocationErrorInfo):Void {
    trace("myMethod - error: " + errorInfo.getException());
}
</code>

```

Method Declaration

Constitution: Methods are declared in classes and interfaces. The method declaration is the part of the method before the curly braces. It consists of the access specifier „public“ or „private“, the optional scoping specifier „static“, the statement „function“, the name of the method, the parameter list and the return type. Interfaces can only declare methods, while classes must also provide implementations for them.

Example of Method Declaration and Implementation in Class:

```

public static function getLogger(loggerName:String):Logger {
    // Implementation omitted
}

```

Example of Method Declaration in Interface:

```

public function isDebugEnabled(Void):Boolean;

```

Line Length: The line length of the method declaration, as well as any other code, should be no longer than 120 characters. A word-wrap should be made somewhere between 80 and 120 characters, when the maximal line length is exceeded. This makes it possible to print the code and to view it with almost any editor. The line after the word-wrap is indented by 8 blank spaces to visually separate it from the method body. (As you can see in the example it is damn hard to have a method declaration that exceeds the maximal line length. If you have such a method you maybe did something else wrong. ;))

Example:

```

public static function
doSomethingVeryVeryVeryInteresting(argumentOne:ReallyInterestingClass,
argumentTwo:ArgumentTwo,
argumentThree:String, argumentFour:Number):String {

```

Method Visibility or Method Access

Public: „public“ is the easiest access specifier. Public methods can be accessed from anywhere in your application.

Private: „private“ is the most restrictive access level. A private member is accessible only to the class in which it is declared and to any sub-classes.

Usage: Hide methods as much as possible. The fewer methods that are public, the cleaner a class is and the easier it will be to test, use and refactor. The public methods that a class exposes will often be only the methods of the interfaces it implements and methods needed for configuration, often getter and setter methods.

Do always declare the visibility of a method even if it is public which is the default case in ActionScript 2 and could simply be neglected. The problem with not declaring the visibility is that developers who read the code must know that the default visibility is public and that it makes the code more difficult to read.

Good Example:

```

public function getLevel(Void):LogLevel;

```

Bad Example:

```
function getLevel(Void):LogLevel;
```

Method Scope

Per Instance: Per instance methods are the default case. You do not have to declare methods as being per instance using any special keyword. Such methods exist per instance. This means that you invoke them on the instance of a class and not on a class itself. You therefore must create an instance of a class first and you can then invoke methods on that instance. Try to use per instance methods as often as possible and avoid per class methods. Per instance methods are much more flexible and open you the door to Design Patterns and key Object Oriented Programming (OOP) concepts like polymorphism.

It is possible to use the „this“ keyword to refer to the instance the method is invoked on.

Example:

```
public function doSomething(Void):Void {  
    trace(„The method ‚doSomething‘ is invoked on instance: " + this);  
}
```

Per Class (Static): Per class methods are also known as static methods. You declare them using the statement „static“. This statement is placed after the visibility and before the „function“ statement. Per class methods exist per class and not per instance. This means that you invoke them directly on the class. It is not possible to invoke them on instances. You reference the method using the class name plus the method name. Only per instance or per class methods declared in the same class as the per class method can invoke it directly by its name without having to specify the class. Do not overuse per class methods. Use them only if they are really appropriate because there are some problems associated with their usage:

- Per class methods can neither be declared nor implemented on interfaces.
- All callers are tied to the class that declares the per class method and to the per class method itself.
- It is not possible to override per class methods.
- Per class methods hold state in per class variables. This means that the state is shared among all callers in the whole application and that the state can be changed from everywhere.

As you can see, using per class methods makes your code less flexible and error-prone when the per class methods hold state. Therefore think twice before using per class methods in your code.

A useful and appropriate per class method is for example the „org.as2lib.env.log.LogManager.getLogger(loggerName:String):Logger“ method. This method is responsible for returning loggers from one and the same repository for every class in the whole application that needs one. It does not really do any functionality but just forwards all the work to the repository that has been set prior to starting the application.

It is not possible to use the „this“ keyword directly in per class methods because using it causes a compiler error. It is nevertheless necessary in some cases to use this keyword to refer to the class the method is declared and implemented on. To do this you can use the „eval“ method as follows: `eval(„th“ + „is“)`.

Example:

```
public static function doSomething(Void):Void {  
    trace(„The static method ‚doSomething‘ is invoked on class: " + eval(„th“ + „is“));  
}
```

Method Name

Formation: A method name starts with a verb. This verb describes in an abstract manner what the method does. It is possibly followed by other words that describe the function of the method more clearly. Take a look at the sections Accessor Methods and Mutator Methods for more information on common verb prefixes.

Good Example:

```
public function loadXmlData(xmlDataSource:String):Void;
```

Bad Example:

```
public function xmlData(xmlDataSource:String):Void;
```

Upper and Lower Case: The first word of a method name is written lower case. Following qualifying words start with upper case letters. When you use abbreviations like XML in the method name, write only the first letter in upper case. If the abbreviation exists only of two letters think about not using it and writing it out. Do for example not use the abbreviation MC for MovieClip in method names.

Good Example:

```
public function loadXmlData(xmlDataSource:String):Void;
```

or:

```
public function createMcLoader(Void):MovieClipLoader;
```

Bad Example:

```
public function loadXMLData(xMLDataSource:String):Void;
```

or:

```
public function createMovieClipLoader(Void):MovieClipLoader;
```

Suffixes: Do not use single suffixes that reference to parameters like „to“, „than“ and „by“ in method names.

Good Example:

```
public function isMoreExplicit(overloadHandler:OverloadHandler):Boolean;
```

or:

```
public function apply(object);
```

or:

```
public function move(x:Number, y:Number):Void;
```

Bad Example:

```
public function isMoreExplicitThan(overloadHandler:OverloadHandler):Boolean;
```

or:

```
public function applyTo(object);
```

or:

```
public function moveBy(x:Number, y:Number):Void;
```

Plural or Singular: Method names are allowed to be plural. That means if you are returning a collection of elements name your method with a word in plural. I do not advocate the use of the collection type in method names. The type is already declared as return type in the method declaration and should not also be in the method name.

Good Example:

```
public function getAllListeners(Void):Array;
```

Bad Example:

```
public function getAllListenerArray(Void):Array;
```

or:

```
public function getListenerArray(Void):Array;
```

or:

```
public function getAllListener(Void):Array;
```

Language: Write method names in American English to avoid a mix of different languages and to ensure that every developer can read the method names properly. Using only English words also guarantees that every alphabetical letter needed is allowed to be part of a method's name.

Statement: A method name should be self-explanatory. To meet this goal do not use abbreviations. Use them only if they are common and you are really sure that the developers who use your code know their meaning. Do not forget to provide the written out form of your abbreviation in the documentation.

You should also qualify the method name with as much information as needed to make it self-explanatory and to leave no space for misinterpretation. Imagine for example a „BeanDefinition“ class. This class is responsible for holding information about beans. This is for example the name of the bean and the class of the bean. While you could name the two accessor methods for these information „getName“ and „getClass“ one could misinterpret them and think „getName“ returns the name of the definition. The same is true for „getClass“. But if you qualify the method names and name them „getBeanName“ and „getBeanClass“, it is almost impossible to misinterpret them, and the four characters you have to write more will not hurt you. This said, do not over-generalize method names, but provide all necessary information. As said above, a method name should be self-explanatory. Take for example a method named „loadXmlData“. This name implies that something is loaded from outside of the flash environment, that this something is data and that this data is encoded in XML format. The fact that XML data is loaded from outside of the flash environment also implies that the loading might fail and that it might be necessary to pay attention to error handling in form of a „null“ return value, an exception or the invocation of an „onError“ event method.

Good Example:

```
public function loadXmlData(xmlDataSource:String):XMLLoaderCallback;
```

Bad Example:

```
public function load(source:String):XMLLoaderCallback;
```

Method Parameter List

Declaration: The parameter list is a comma delimited list that is declared after the method name. The list starts with an opening brace „(“ and ends with a closing brace „)“. The opening brace is put directly after the method name, with no blank space in-between. The individual parameters are declared between the braces and separated by commas. There is a blank space after every comma, that is before every new parameter. Every parameter has a name and a type. The type is declared using ActionScript's 2 strict data typing post-colon syntax, „myParameter:MyType“.

Good Example:

```
public function log(message:String, level:LogLevel):Void;
```

Bad Example:

```
public function log ( message,level ):Void;
```

or:

```
public function log(message : String,level : LogLevel) : Void;
```

Name: The name of a parameter should be descriptive and self-explanatory. It should also specify the context of the parameter.

Good Example:

```
public function loadXmlData(xmlDataSource:String):Void;
```

Bad Example:

```
public function loadXmlData(source:String):Void;
```

You may use the prefix „new“ for parameters of setter methods.

Example:

```
public function setName(newName:String):Void;
```

As with method names the name of a parameter should be written in American English to avoid a mix of different languages and assure that every developer in your team can read the method names properly.

Usage: Due to the sloppy nature and some bugs of ActionScript 2 regarding parameters we must be careful of how to use them.

A method normally declares all parameters and their types in its parameter list.

Example:

```
/**
 * [Main Description]
 *
 * @param level the level to check whether this logger is enabled for it
 * @return true if this logger is enabled for the passed-in {@code level}
 * else false
 */
public function isEnabled(level:LogLevel):Boolean;
```

But if you expect a parameter of any type, do not use „Object“ as type but leave the parameter type empty. Do this because the Macromedia compiler thinks that instances casted to interfaces are not of type „Object“. In this case you may also think about using a marker interface as type. This is an interface that does not declare any methods.

Example:

```
/**
 * [Main Description]
 *
 * @param target the target object to stringify
 * @return the string representation of the passed-in {@code target}
 * object
 */
public function stringify(target):String;
```

If the method expects no parameters do not just leave the parameter list empty, because this could also imply that the method expects an unknown number of parameters, but specify „Void“ as parameter.

Example:

```
/**
 * [Main Description]
 *
 * @return the name of this bean
 */
public function getBeanName(Void):String;
```

If the method expects an unknown number of parameters, including zero parameters, leave the parameter list empty. This is for example the case with an overloading method that forwards the functionality to other methods depending on the passed-in parameters.

Example Normal Method:

```
/**
 * [Main Description]
 *
 * @param .. any number of values, that are of the expected type that was
 * specified on construction, to concat with this array
 * @return an array that contains the values of this array as well as the
 * passed-in ones
 */
public function concat():TypedArray;
```

Example Overload Method:

```
/**
 * @overload getBeanByName
 * @overload getBeanByNameAndType
 */
public function getBean();

/**
 * [Main Description]
 *
 * @param beanName the name of the bean to return
 * @return the bean corresponding to the passed-in {@code beanName}
 */
public function getBeanByName(beanName:String);
```

```

/**
 * [Main Description]
 *
 * @param beanName the name of the bean to return
 * @param beanType the expected type of the bean to return
 * @return the bean corresponding to the passed-in {@code beanName} that
 * is of the expected {@code beanType}
 */
public function getBeanByNameAndType(beanName:String, beanType:Function);

```

If the method expects an unknown number of parameters, but at least one declare one parameter in the parameter list. Also declare the type of the parameter if known.

Example with Known Type:

```

/**
 * [Main Description]
 *
 * @param neededParameter [description]
 * @param .. any number of parameters that are of type {@code String}
 * and [further description]
 */
public function doSomething(neededParameter:String):Void;

```

Example with Unknown Type:

```

/**
 * [Main Description]
 *
 * @param value the new value, that is of the expected type specified on
 * construction, to add to the end of this array
 * @param .. any number of values, that are of the expected type specified
 * on construction, to add to the end of this array
 * @return the new length of this array
 */
public function push(value):Number;

```

Do not forget to comment all of the above special cases clearly in your method documentation, as it is done in the examples.

Method Return Type

Always declare the return type of the method. If the method does not return any value declare „Void“ as return type.

Example:

```

/**
 * [Main Description]
 *
 * @return the name of this bean
 */
public function getBeanName(Void):String;

/**
 * [Main Description]
 *
 * @param beanName the new name of this bean
 */
public function setBeanName(beanName:String):Void;

```

In case the return type can be any type do not use „Object“ as return type but leave the return type signature empty. Do this because the Macromedia compiler thinks that instances casted to interfaces are not of type „Object“.

Example:

```

/**
 * [Main Description]
 *
 * @return the message object to log
 */
public function getMessage(Void);

```

Method Body or Method Implementation

Constitution: The method body is implemented using curly braces. The opening curly brace is on the same line as the method declaration. The curly brace is separated from the declaration by a blank space. The first line of code of the implementation is on the next line and indented by four blank spaces. The last line contains only the closing curly brace that is aligned to the beginning of the method declaration.

Good Example:

```
public function getBeanName(Void):String {
    return beanName;
}
```

Bad Example:

```
public function getBeanName(Void):String{
return beanName;
}
```

or:

```
public function getBeanName(Void):String {
    return beanName;
}
```

or:

```
public function getBeanName(Void):String
{
    return beanName;
}
```

Error Handling: Every method should implement its own error handling. That means it must be possible to invoke every method, public or private, directly and that special cases like „null“ parameters must be handled correctly even if dependent methods were overridden by a sub-class. Do thus not make your error handling dependent on other methods.

„super“: Within method bodies you can use the „super“ operator to explicitly invoke methods of super-classes. While it is possible to use „super“ every time you invoke methods of super-classes, use it only if you have overridden a super-class’s method and you want to invoke the overridden method.

Line Length: As with the method declaration, the line length should be limited to 120 characters. If your code exceeds this maximal line length, make a word-wrap somewhere between 80 and 120 characters. This makes it possible to print the code and to view it with most editors. The code after the word-wrap is indented by 8 blank spaces to visually separate it from following code and to affiliation to the preceding line.

Example:

```
throw new IllegalArgumentException("The types of the arguments [" + arguments + "] must
match one of the two choices.",
    this, arguments);
```

Method Response

Return Value: The method can response to an invocation by returning a value. This value can for example be the result of a computation or can inform the caller of the success or failure of the invocation. Use return values for the latter case only in low-level APIs. It is recommended to use exceptions in high-level APIs because they provide much more information about what exactly went wrong and where it went wrong.

A method returns a value using the „return“ statement. This statement is followed by a blank space and the value to return. If the value is computed using operators then put the computation into parenthesis. Do not use parenthesis if you create an instance to return using the „new“ operator.

Good Example:

```
return size;
```

```

or:
return (size != null ? size : defaultSize);
or:
return new MyClass();
Bad Example:
return (size);
or:
return size != null ? size : defaultSize;
or:
return (new MyClass());
or:
return(size != null ? size : defaultSize);

```

Exceptions: Throw exceptions to indicate that something went wrong. Distinguish between normal and fatal exceptions. Normal exceptions are likely to be caught and handled by the application, while fatal exceptions indicate programming errors that must be handled by application developers. Fatal exceptions are for example „org.as2lib.env.except.IllegalArgumentException“ that is thrown on illegal arguments and „org.as2lib.env.except.AbstractOperationException“ that is thrown if the developer forgot to overwrite an abstract method. A normal exception is „org.as2lib.io.conn.core.client.UnknownServiceException“ that is thrown if a service is not available right now. The application can catch this exception and for example try to reconnect after waiting one second. Exceptions are thrown using the „throw“ statement. Do not use parenthesis after the „throw“ statement.

```

Good Example:
throw new MyException(„A descriptive message.“, this, arguments);
Bad Example:
throw(new MyException(„A descriptive message.“, this, arguments));

```

An error handling article will be available soon.

Logging: A method may also response by logging. This kind of response differs from the two above in that it is only intended for developers and does not affect the execution of the application. The method may response at error or fatal level if it caught a normal exception or fatal exception. It may log a warning if it is in a state that should not actually occur or something unexpected happened that poses no problem to the execution of the application. The method logs at info and debug level to inform developers of what it is going to do. All this logging helps developers to easily track down errors of the application. An article about logging will be available soon.

Constructor Method

A constructor is responsible for creating and initializing instances of a class. Constructors must have the same name as the class that declares it. Constructor names do thus not start with a verb and do also not start with a lower case letter but with an upper case one, like class names. Constructors do also not declare a return type, because the return type is always the class that declares the constructor.

```

Example:
public function ClassInfo(clazz:Function, name:String, package:PackageInfo) {
    this.clazz = clazz;
    this.name = name;
    this.package = package;
}

```

You do not have to declare a constructor for a class. If you do not do it the compiler automatically creates an empty constructor. It is suggested to always declare a constructor for every class you create whether it performs any functionality or not. Declaring the

constructor makes you think about which parameters the super-class's constructor expects and makes you implement your constructor the way that the expected parameters are passed-to the super-class's constructor. Otherwise the compiler creates a new constructor that expects no parameters and all parameters are „undefined“ when the super-class's constructor is invoked. Note that the super-class's constructor is always invoked whether you invoke it by hand using „super“ in the constructor body or not. But with using „super“ by hand you have the possibility to pass parameters to the super class's constructor.

Example with Super-Class's Constructor that Expects No Parameters:

```
public function ClassAlgorithm(Void) {  
}
```

Example with Super-Class's Constructor that Expects Four Parameters:

```
public function ConstructorInfo(constructor:Function, declaringClass:ClassInfo) {  
    super (NAME, constructor, declaringClass, false);  
}
```

Accessor Methods

Naming: Accessor methods are usually named after what they return.

Getter Methods: Getter methods are accessor methods that return non-boolean values. Their names start with the verb „get“. These methods normally do not do much computation but just look up a variable so they can be assumed to execute fast.

Example:

```
public function getBeanName(Void):String {  
    return this.beanName;  
}
```

Test Methods: Test methods are accessor methods that return boolean values. These methods normally give information about the object by testing its state or indicate whether some specific operations shall take place. Common prefixes are forms of to be like „is“ and „was“, „has“, „can“, „should“ and „will“.

Example:

```
public function isRootPackage(Void):Boolean {  
    return (getParentPackage() == null);  
}
```

or:

```
public function hasListeners(Void):Boolean {  
    return (this.listeners.length > 0);  
}
```

or:

```
public function canEvaluate(Void):Boolean;  
public function shouldAbort(Void):Boolean;  
public function willTerminate(Void):Boolean;
```

Creation Methods: Creation methods are accessor methods that create and return a new instance. Use the verb „create“ for their names. Do not use „new“ because it is not a verb and because it is easy to mistake it for a real instantiation of a class in a method call.

Good Example:

```
public function createLogger(loggerName:String):Logger {  
    return new SimpleLogger(loggerName);  
}
```

Bad Example:

```
public function newLogger(loggerName:String):Logger {  
    return new SimpleLogger(loggerName);  
}
```

Search Methods: Search methods are accessor methods that search for something and return either the result or return „null“ or throw an exception. Use the verb „find“ for

the names of these methods. Search methods can be assumed to need some time for execution.

Example:

```
public function findBeansOfType(beanType:Function):Array {
    var result:Array = new Array();
    for (var i:Number = 0; i < this.beans.length; i++) {
        if (beans[i] instanceof beanType) {
            result.push(beans[i]);
        }
    }
    if (getParentBeanFactory()) {
        result = result.concat(getParentBeanFactory().findBeansOfType(beanType));
    }
    return result;
}
```

Computation Methods: Computation methods are accessor methods that compute something and return the result. They are normally used to do arithmetic calculations. You prefix them with the verb „compute“.

Example:

```
public function computeSomething(parameter1:Number, parameter2:String):Number;
```

Generation Methods: Generation methods are similar to computation methods. The difference is that they are used for any kind of generation, while computation methods are mostly used for arithmetic calculations. Generation methods are prefixed with the verb „generate“.

Example:

```
public function generateResponseServiceUrl(serviceUrl:String, methodName:String):String;
```

Conversion Methods: Conversion methods convert objects and return the result. The most common conversion method is „toString“ that creates a string representation of the object it is invoked on. As you can see conversion methods are prefixed with the adverb „to“.

Example:

```
public function toArray(Void):Array {
    return this.data.concat();
}
or:
public function toString(Void):String {
    return („[type " + ReflectUtil.getTypeName(this) + "]"");
}
```

Mutator Methods

Naming: Mutator methods are usually named after what they mutate or after what they do.

Setter Methods: Setter methods are mutator methods that simply set an instance or class variable. They can thus be supposed to execute rather fast. Prefix their method name with the verb „set“ and their parameter’s name with the adjective „new“. Setter methods can be invoked as often as you please.

Example:

```
public function setName(newName:String):Void {
    this.name = newName;
}
```

Initialization Methods: Initialization methods are mutator methods that initialize or establish an object or a concept, for example a whole instance or just a variable. Prefix their names with the verbs „init“ or „initialize“ or „setup“. Initialization methods can normally only be invoked ones. For example a method with name „initName“ can be supposed to do the same as a simple setter method with the difference that it for example

throws an „org.as2lib.env.except.IllegalStateException“ if invoked more than ones.

Example:

```
public function initName(name:String):Void {
    if (!name) {
        throw new IllegalArgumentException(„Parameter ‚name‘ [“ + name + „] must not be
        ,null‘, ‚undefined‘ or an empty string.“, this, arguments);
    }
    if (this.name != null) {
        throw new IllegalStateException(„The name [„ + this.name + „] has already been
        initialized.“, this, arguments);
    }
    this.name = name;
}
```

Collection Methods: Collection methods are mutator methods that add values to, remove values from or get collections. Do not indicate a specific ordering of the internal collection with your collection methods’ prefixes if the ordering is not necessary. Without a specific order you have more freedoms of how to implement the method. Prefix a collection method that assigns a value to a key „put“. This term does not indicate any special value ordering.

Example:

```
public function putLogger(loggerName:String, logger:Logger):Void;
```

Use the prefix „add“ if values are just added to the collection. This term does also not indicate any special value ordering.

Example:

```
public function addListener(listener:Listener):Void;
```

Use the prefix „remove“ to remove values from the collection.

Example:

```
public function removeListener(listener:Listener):Void;
```

The prefix „push“ is used if values are added to the end of a collection and „pop“ if values are removed from the end of a collection. Note that these two prefixes are used in conjunction and ensure the last-in, first-out policy, that gets implemented by stacks.

Example:

```
public function push(value):Void;
```

or:

```
public function pop(Void);
```

If values are removed from the beginning of a collection use the prefix „dequeue“. Use the prefix „enqueue“ in conjunction with this method to add values to the end of the queue. These two prefixes are, as already indicatd, used in conjunction. They ensure the first-in, first-out policy, that gets implemented by queues.

Example:

```
public function enqueue(value):Void;
```

or:

```
public function dequeue(Void);
```

Use the prefix „getAll“ if all values of a collection are returned. Try to return the values in a general collection type like an array, regardless of the internal type. Almost all collections provide functionalities to convert them into arrays. This gives your more freedoms for your internal implementation. Most developers are also more familiar with working with an array than with a list or a queue. Do never return the internal collection itself, but return a copy of it or make it immutable before you return it. Otherwise it would be possible to edit the internal collection from outside, which easily leads to unexpected behavior.

Example:

```
public function getAllListeners(Void):Array {
    return this.listeners.toArray().concat();
}
```

Abstract Methods

Introduction: Abstract methods are methods that do not provide an implementation but only a declaration. You can compare them with method declarations in an interface. The difference is that abstract methods are declared in abstract classes, that are classes that implement some methods of for example an interface and leave some other methods abstract. The concrete sub-class is then responsible for implementing these abstract methods.

Declaration: Abstract methods are not supported by ActionScript 2. But you can simply throw an exception of type

„org.as2lib.env.except.AbstractOperationException“ to mark the method as being abstract and to get a run-time exception if you forgot to overwrite the abstract method. Because this is actually only a work-around do only use it if their usage does really improve your code.

If the abstract method declares a return type you must return something to please the compiler, even if the „return“ clause is not reachable due to the „throw“ clause. In such a case do always return „null“, as in the example.

Example:

```
private function doLoad(dataSource:String):String {
    throw new AbstractOperationException(„This method is marked as abstract and must be
    overridden.“, this, arguments);
    return null;
}
```

Naming: When an abstract class implements a method that does some general operations and forwards others to an abstract method and, you cannot find a name that describes the functionality of that abstract method more clearly than the name of the forwarding method, name this abstract method like the forwarding method with the prefix „do“.

Example:

```
public function load(dataSource:String):String {
    if (loadedFiles.contains(dataSource)) {
        return loadedFiles.get(dataSource);
    }
    var result:String = doLoad(dataSource);
    loadedFiles.put(dataSource, result);
    return result;
}

private function doLoad(dataSource:String):String {
    throw new AbstractOperationException(„This method is marked as abstract and must be
    overridden.“, this, arguments);
}
```

Overload Methods

Introduction: Overloading is not supported by ActionScript 2. That means it is not possible to declare multiple methods with the same name that only differ in their parameter list or return type.

Naming: A naming convention is needed that gives information about the actual functionality and about the specific way of performing this functionality. This is done using the actual name of the method plus the adverb „by“ or the prepositions „for“ or „of“ plus the base names of the parameters, without the context part, separated by the conjunction „And“. If the method that is forwarded to takes no parameters use „Void“ as parameter name, „getMethodsByVoid(Void):Array“. Do not qualify the actual

name of the most common method by „by“, „for“ or „of“ or use the actual name for the method that forwards to the qualified methods.

Forwarding: Forwarding can be done depending on the length of the parameter list or the types of the parameters or both using if-else statements. While this is possible it is awkward to implement for every forwarding method and it is likely that special cases are ignored. I thus suggest you to use the As2lib Overloading API which does all the overloading for you and provides clean error handling.

Visibility or Access Specifier: The forwarding method as well as the qualified methods that are forwarded to have all the same access specifier. That means if the forwarding method is public, the qualified methods are public too. This ensures that each method can be invoked directly without the overhead produced by the forwarding. Note that with overloaded constructors the qualified constructors are always private, because otherwise they could be invoked directly.

Parameter List: The forwarding method declares no parameters in its parameter list.

Return Type: The forwarding method declares a return type if all methods it forwards to also declare the same return type. If the return type of the methods it forwards to differ declare no return type.

Example: If the method to overload has the name „getBean“ and the method can either be invoked using the name and the expected type of the bean or using only the bean’s name, name your forwarding method „getBean“ and the two other methods „getBeanByNameAndType“ and „getBeanByName“ respectively.

Example using the As2lib Overloading API

```
/**
 * @overload getBeanByName
 * @overload getBeanByNameAndType
 */
public function getBean() {
    var o:Overload = new Overload(this);
    o.addHandler([String], getBeanByName);
    o.addHandler([String, Function], getBeanByNameAndType);
    return o.forward(arguments);
}

/**
 * [Main Description]
 *
 * @param beanName the name of the bean to return
 * @return the bean corresponding to the passed-in {@code beanName}
 */
public function getBeanByName(beanName:String) {
    ...
}

/**
 * [Main Description]
 *
 * @param beanName the name of the bean to return
 * @param beanType the expected type of the bean to return
 * @return the bean corresponding to the passed-in {@code beanName} that
 * is of the expected {@code beanType}
 */
public function getBeanByNameAndType(beanName:String, beanType:Function) {
    ...
}
```

Further Example:

```
public function getBeansOfType(beanType:Function):Array;
```

OR:

```
public function getBeanNamesForType(beanType:Function):Array;
```

Event Methods

Event Methods are invoked when an event is broadcasted. They capture events. Event methods are implemented on listeners, that can be registered at broadcasters. A common prefix for event methods is „on“. Macromedia itself is following this convention as you can for example see at all the MovieClip events like „onEnterFrame“ and „onRelease“. An article about event handling will be available soon.

Anonymous Methods

Introduction: Avoid the use of anonymous methods as often as possible. There are some major drawbacks with their usage. The compiler does for example not verify the parameters and the return type in the method declaration, which may lead to huge problems when the type signature of the method declaration is changed in the declaring class and you do not get a compiler error for your anonymous methods.

Declaration: Anonymous methods are simply functions created at run-time that are assigned to a variable.

Example:

```
handler.invoke = function(proxy, methodName:String, args:Array) {  
    ...  
};
```

or:

```
var invoke:Function = function(proxy, methodName:String, args:Array) {  
    ...  
};
```

Note the semicolon at the end of the variable initialization.

Access: As said above anonymous methods are functions that are assigned to a variable. You therefore access them using the variable's name. They are invoked like any other method.

Example:

```
handler.invoke(myProxy, „doSomething“, [„parameter1“, 2]);
```

or:

```
invoke(myProxy, „doSomething“, [„parameter1“, 2]);
```

Usage: Anonymous methods are mostly used for event handling and anonymous classes. They can be assigned for example to events on movieclips like „onEnterFrame“.

Example:

```
myMovieClip.onEnterFrame = function(Void):Void {  
    ...  
};
```

Yet a much cleaner way is to extend the MovieClip class and declare the instance method „onEnterFrame“ in this class.

Example:

```
class org.simonwacker.myapplication.view.MyMovieClip extends MovieClip {  
    ...  
    private function onEnterFrame(Void):Void {  
        ...  
    }  
    ...  
}
```

As I said they are also used for anonymous classes. Take the „org.as2lib.io.conn.core.event.MethodInvocationReturnListener“ for example. It prescribes that sub-classes must implement exactly one method „onReturn“. Let us say you call six remote methods and the return value of every of these methods must be handled in a different way. In such a case you may consider to use anonymous implementations of the „MethodInvocationReturnListener“ class. You simply instantiate the interface (note that this does not work in Flex directly) and

initialize the „onReturn“ method with an anonymous method for every of the six remote method invocations. This saves you from creating six classes that maybe only do some trivial operations.

Example (not Flex compatible):

```
var returnListener:MethodInvocationReturnListener = new MethodInvocationReturnListener();
returnListener.onReturn = function(returnInfo:MethodInvocationReturnInfo):Void {
    ...
};
```

Example (Flex compatible):

```
var temporaryReturnListener = new Object();
temporaryReturnListener.__proto__ = MethodInvocationReturnListener.prototype;
var returnListener:MethodInvocationReturnListener = temporaryReturnListener;
returnListener.onReturn = function(returnInfo:MethodInvocationReturnInfo):Void {
    ...
};
```

„toString“ Method

Every class should implement a „toString“ method that returns at least the name of the class. This helps alot with diagnostics and is especially helpful in generating log messages, that help in debugging. It is even better if the „toString“ method returns the name of the class as well as its current state. A common convention of how to generate string representations of classes is the following:

Minimal Syntax:

```
[type fullyQualifiedClassName]
```

Syntax with State:

```
[type fullyQualifiedClassName
  variable1: value1
  variable2: value2
]
```

You can extend the class „BasicClass“ of the As2lib Framework to inherit a „toString“ method that returns the name of your class, as in the first example. If you want to do the stringification on your own the As2lib Reflection API might help you, especially the light-weight „org.as2lib.env.reflect.ReflectUtil“ class.

Method Call

When calling a method there is no blank space between the method name and the parenthesis. There is also neither a blank space between the opening brace and the first parameter nor between the last parameter and the closing brace. Parameters are separated by commas. There is a blank space after every comma but not before it.

Example:

```
myXMLLoader.loadXmlData(„myXmlFile“, myResponseListener);
```

If the parameters are too long or if there are too many parameters, build parameter groups and put each group on its own line. Indent each line to align with the above parameter group.

Example:

```
myXMLLoader.loadXmlData(„myXmlFile“, myResponseListener,
                        myTimeout, myAlternativeXmlFile,
                        myUnnecessaryFlag);
```