XINS

JAVA CODING CONVENTIONS

Version 1.0, July 2006

Table of Contents

1. Introduction

1.1 About this document

This document presents the Java coding guidelines for the XINS project.

It is a known fact that most time spent on software is not in the initial writing, but in the maintenance afterwards. So it is extremely important that code is readable. A consistent coding style will help in this, allowing the reader to quickly interpret what he sees. Also, code conventions can help avoiding programming errors by steering away from certain common mistakes.

Altogether, this style guide is a tool for programmers to improve maintainability and quality of the code they produce.

This document starts off by defining some important general principles, in chapter 2. These are followed by a number of chapters that define guidelines for different aspects of Java source code. Finally, chapter 10 gives some examples of properly formatted Java source code.

If you have any questions or suggestions related to this document, please direct these to the XINS Users Mailing List, at xins-users@lists.sourceforge.net.

1.2 Assumptions

This document assumes that CVS is used as the version management system for source files.

English is used as the language for both comments and naming.

1.3 References

While writing this coding convention, the following documents were used as references:

- *Code Conventions for the Java Programming Language* by Sun Microsystems.
  http://java.sun.com/docs/codeconv/
- *Tips for Maintainable Java Code* by Rolf Howarth.
  http://www.squarebox.co.uk/download/javatips.html
- *C-style – Standards and Guidelines* by David Straker.
  ISBN: 0131168983
- *The Elements of Java Style* by Allen Vermeulen and others.
  ISBN: 0521777682

1.4 Terminology

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in RFC 2119. See http://www.faqs.org/rfcs/rfc2119.html.

In this document, the term *module* refers to a collection of grouped together Java source files, typically coming with some additional files such as unit tests and documentation.

Where “he” is written, it should be interpreted as “he or she”.

2. General principles

2.1 Adhere to the style of the original

When modifying existing software, stick to the style of the original source code. Combining different styles within a single source file is confusing and makes it difficult to read and understand.

Rewriting old code just to apply a different coding style may introduce bugs and needs to be approached with great care. If you really want to change the style, make sure you have automated test cases with good coverage before you start making changes to the code.

2.2 Be predictable

The *Principle of Least Astonishment* states:

> "*When two elements of an interface conflict or are ambiguous, the behavior should be that which will least surprise the human user or programmer at the time the conflict arises, because the least surprising behavior will usually be the correct one*"

This rule applies both to the source code self and to the functionality it implements. The more obvious the choices a programmer makes are, the less effort if required from the reader or user of the software. And vice versa.

2.3 Keep it simple

Simplicity is key to building large yet maintainable systems. Build simple classes and methods. Apply the KISSSS principle:

> *Keep It Short, Simple, Small, and Self-contained*

Focus on high cohesion *within* an item and low coupling *between* items.

2.4 Avoid premature optimization

While designing a system, course-grained, ensure that the design by itself is not likely to become a bottleneck.

However, fine-grained performance optimization should be done at the end of the software development cycle, while profiling, if at all. Do not get distracted by the temptation of optimizing your code too early. As Donald Knuth said:

> "*Premature optimization is the root of all evil (or at least most of it) in programming*"

2.5 Separate concerns

Giving each class and method a specific and limited function will help readers gain a good understanding of the software system. In documentation, present the design in a top-to-bottom manner, so readers can drill down to the information they want.

Systems that do not apply the *Principle of Separation of Concerns* may require the source code reader to read *all* code in order to understand only *one* piece of functionality.

2.6 Be consistent

Similar tasks should be implemented in a similar fashion. Create and apply standards as much as possible. Refer to *Design Patterns* when discussing and documenting designs.

2.7 Write robust code

Make sure your code completely documents how behavior is affected by state and input parameters. Document all exceptions.

Do not hide errors. If you do not know how to let your code handle an error properly, do not handle it but instead, throw it so higher level code has a chance to handle it. You may want to log the error.

2.8 Document and justify deviations

No standard is perfect and no guideline is universally applicable. However, before deviating from a guideline, be sure you understand why the rule was defined in the way it is.

Then if you are sure the rule should not be applied in your case, document exactly what you do differently, and why.

If you want to deviate from a rule that is marked as “MUST” or “REQUIRED” then please discuss this with the other XINS project members first. It may be appropriate to loosen, change or remove the rule.

2.9 Communicate with humans

Discuss your approach with other people. If possible, have somebody else review your code. Consider doing *pair programming*.

3. File organization

3.1 Directory structure

All source files SHOULD be organized in directory structures. The lowest part of the directory hierarchy SHOULD map directly to the package names for the code in those directories. One of the higher-level directories SHOULD be called `src` (for "source"). Intermediate directories SHOULD group common source files.

Example of a directory name for a package named `org.xins.common`:

```
xins/src/java-common/org/xins/common/
```

Here `xins` is a directory for the whole project. Within that directory, there is one directory for all source files. In there, there is a separation per category of source files. One of these categories is called `java-common`, with source files for different packages, including the package named `org.xins.common`.

3.2 Files accompanying the source code

3.2.1 COPYRIGHT

Each module containing source code MUST contain a text file named `COPYRIGHT` in the top directory. This file SHOULD have the following content:

```
Copyright years, Orange Nederland Breedband B.V.
All rights reserved.

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this
   list of conditions and the following disclaimer.

2. Redistributions in binary form must reproduce the above copyright notice,
   this list of conditions and the following disclaimer in the documentation
   and/or other materials provided with the distribution.

3. Neither the name of the copyright holder nor the names of its contributors
   may be used to endorse or promote products derived from this software
   without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDER AND CONTRIBUTORS "AS IS"
AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE
DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE
FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR
SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER
CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY,
OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE
OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
```

where "*license*" needs to be replaced with the actual license text and "*years*" needs to be replaced by the actual copyright years, for example "2003-2006".

Note that when code is submitted for inclusion in XINS, the author will need to agree that the source code is placed under this license.

3.2.2 README

Each module containing source code MUST contain a text file named `README` in the top directory. This file MUST at least contain the following, in order:

1. Name or names for the module, both friendly names and technical ones (REQUIRED)
2. A description of the module, including an explanation of the responsibilities and/or purposes (REQUIRED)
3. A reference to the main web page for the module (OPTIONAL)
4. The owner and authors of the module, typically “Orange Nederland Breedband B.V.” (REQUIRED)
5. References to other locations with relevant information (OPTIONAL)
6. A reference to the COPYRIGHT file (REQUIRED)
7. An `$Id$` tag, for processing by CVS (REQUIRED)

Example of a valid README file:

```
xins-fume (XINS File Upload Modular Extension)

This module implements a generic file upload mechanism for XINS-based
applications. All code is in or below the package org.xins.ext.fume.

By Cap Gemini, for Orange Nederland Breedband B.V.

See:
http://www.xins.org/ext/fume/

For copyright and license information, see the COPYRIGHT file.

$Id$
```

4. File structure

This chapter defines how individual Java source files should be structured.

4.1 High-level overview

A Java source file MUST contain the following sections, in order:

1. The REQUIRED file header; see section 4.2.
2. The REQUIRED package declaration; see section 7.3.
3. Any (OPTIONAL) import statements, prefixed by an empty line; see section 8.2.
4. The REQUIRED outer class comment, prefixed by an empty line; see section 6.7.5.
5. The REQUIRED definition of the (outer) class.

There MUST NOT be more than one outer class per Java source file.

4.2 File header

Each Java source file MUST start with a header in the following form:

```
/*
 * $Id$
 *
 * Copyright years Orange Nederland Breedband B.V.
 * See the COPYRIGHT file for redistribution and use restrictions.
 */
```

where *years* MUST be replaced with the applicable copyright years, for example “2006-2007”.

When this file is stored in CVS, then CVS will replace the tag `$Id$` by a text string containing the file name, the version number, a timestamp, the last committer and a state tag. For example:

```
$Id: Main.java,v 1.54 2005/12/30 23:55:12 johndoe Exp $
```

As new versions of the file are committed to CVS, the `$Id$` tag will automatically be updated.

See chapter 10 for some examples.

5. Naming conventions

5.1 Naming in general

Identifiers SHOULD be short yet familiar and explanatory; use whole words.

One-letter identifiers SHOULD NOT be used for entities other than local variables and even then they SHOULD be used sparsely and only for primary data types. Realize that shorter terms such as “n” can be extremely hard to search for.

When a capitalized word or acronym, such as “HTTP” or “JDBC” is included in a name, it should be formatted as if it were a normal word. For example:

```
// Copy reference in class field to instance field
_httpServiceCaller = HTTP_SERVER_CALLER;
```

Names SHOULD NOT include acronyms and abbreviations, unless the abbreviation is much more widely used than the long form, such as “URL” and “HTML” for example.

5.2 Packages

Package names MUST be all lower case, starting with the inverse of the applicable domain name.

Package names for closed source code MUST start with “`nl.orange.bb.`” Examples include:

```
nl.orange.bb.common
nl.orange.bb.common.text
nl.orange.bb.frontends.modgui
```

For open source code a package name SHOULD be used that does not refer to the company, but instead to the domain name for the open source project. For example:

```
org.xins.common
org.xins.common.text
org.xmlenc
```

5.3 Classes and interfaces

A class SHOULD have one identifiable responsibility. The class name SHOULD indicate clearly what that responsibility is.

Class names SHOULD be nouns, in mixed case with the first letter of each internal word capitalized.

The same rules apply for interfaces.

5.4 Methods

A method SHOULD have one identifiable task. The method name SHOULD indicate clearly what that task is. Use the JavaBeans™ naming convention for property accessor methods.

Within interfaces, the method declarations MUST NOT contain access modifiers. See section 10.1 for an example.

5.5 Fields

5.5.1 Class fields

Class field names MUST start with an uppercase letter. The name SHALL only contain uppercase letters, digits and underscores. Digits SHOULD be avoided. Words SHOULD be separated using an underscore.

Example of valid class field names:

1. `SINGLETON`
2. `INSTANCE_COUNT_LOCK`
3. `ISO_9700_SCORE`
4. `HTTP_SERVICE_CALLER`

5.5.2 Instance fields

Instance field names MUST start with an underscore followed by a lower-case letter. The name SHALL only contain letters, digits and underscores. They SHOULD follow the Hungarian notation.

Digits SHOULD be avoided, as well as underscores (except at the first position).

Examples of valid instance field names:

1. `_items`
2. `_itemCount`
3. `_iso9700Score`
4. `_httpServiceCaller`

5.6 Local variables

Local variable names MUST start with a lower-case letter. The name SHALL only contain letters, digits and underscores. They should follow the Hungarian notation.

Digits SHOULD be avoided, as well as underscores (except at the first position).

Examples of valid instance field names:

1. `items`
2. `itemCount`
3. `iso9700Score`
4. `httpServiceCaller`

Caught exceptions (in a `catch` block) SHOULD be called `exception`. For example:

```
// Connect to the other end
try {
   connect();

// Connection time-out
} catch (ConnectException exception) {
   // TODO: handle

// Socket time-out
} catch (SocketException exception) {
   // TODO: handle
}
```

6. Comments

This chapter defines rules for the usage of comments, their placement and their formatting. Also, it gives some guidelines regarding the wording to use.

6.1 General rules

Code and comments are equally important. Both SHOULD be kept in sync at all times.

Comments SHOULD be written in the active form.

Needless words SHOULD be skipped. For example, never start the first sentence of a documentation comment with “This” or “The”.

6.2 C-style block comments

There are several styles of comments. C-style block comments start with:

```
/*
```

and end with:

```
*/
```

C-style block comments MUST NOT be used except for commenting out blocks of code.

6.3 Commenting out code

To temporarily disable sections of code, C-style comments MAY be used. For example:

```
// Change the internal data
Freuqncy oldFrequency = _frequency;
_frequency = newFrequency;

/*
// Notify all registered listeners
notifyListeners(NEW_FREQUENCY_EVENT, oldFrequency, newFrequency);
*/

// Lastly, notify the parent
_parent.childNewFrequency(this, oldFrequency, newFrequency);
```

Commented out code SHOULD NOT be committed to the version management system.

6.4 Empty constructors and methods

All constructors and methods that are intentionally left empty SHOULD contain only one, properly indented, line, as follows:

```
// empty
```

This clearly shows that implementation is not just forgotten. For example:

```
protected DummyTarget() {
   // empty
}
```

6.5 Section demarcation comments

An *outer class* definition MUST be divided in the following sections, in order:

1. Class fields – All fields marked as `static`.
2. Class functions – All `static` methods.
3. Constructors.
4. Fields – All instance fields (so not `static`).
5. Methods – All instance methods (so not `static`).
6. Inner classes – All inner classes.

Each non-empty section MUST be prefixed by a demarcation comment. Outer classes SHOULD have demarcation comments for all non-empty sections.

A demarcation comment MUST respect the applicable indentation level. It MUST consist of three lines:

1.The first line MUST start with two slash characters, followed by hyphens until the maximum line length is reached.

2.The middle line MUST contain the title of the section that follows.

3.The third line MUST equal the first one.

See section 10.3 for an example of an outer class definition that includes properly formatted demarcation comments.

6.6 Inline comments

Inline comments are comments that are placed within a block of code. Inline comments start with:

```
//
```

and end at the end of the line.

All inline comments MUST be on separate lines, not behind a statement. They MUST be properly indented. There MUST be at least one empty line before the first comment line.

6.6.1 Implementation comments

Implementation comments are inline comments that describe certain aspects of the implementation. Typically, they summarize and explain the flow within in a block of code.

For example:

```
public ProgrammingException(String    subjectClass,
                            String    subjectMethod,
                            Throwable cause) {

   // Call superconstructor with a constructed message
   super(createMessage(subjectClass, subjectMethod,
                       detail,       cause));

   // Register the cause for this exception
   ExceptionUtils.setCause(this, cause);

   // Store the rest of the information in fields
   _subjectClass    = subjectClass;
   _subjectMethod   = subjectMethod;
}
```

Implementation comments SHOULD start with a verb.

6.6.2 Note comments

Note comments are inline comments that give background information or alert readers. The following types of note comments are distinguished:

- NOTE – Background information.
- XXX – Alert that indicates a potential problem.
- TODO – Indication of a critical issue that must be resolved.

Production code MUST NOT contain any TODO comments anymore.

Here is an example of some note comments:

```
// NOTE: The behavior of this class has been slightly redefined in XINS
//       1.1. In XINS 1.0, the name for a DataElement was a combination of
//       the namespace prefix and the local name. In XINS 1.1, the name is
//       just the local name. Since XINS 1.0 did not support XML Namespaces
//       yet, this is not considered an incompatibility.

// TODO: Check the validity of the 'local name' since it can currently be
//       anything, e.g. an empty string "" or some text including spaces,
//       such as " bla dee bla"
```

Note comments that pertain to a class or to a field SHOULD be inserted at the very top of the class body. Those that pertain to a single method SHOULD be inserted either at the very top of the class body or at the very top of the method body.

Example of a note comment that pertains to a method:

```
/**
 * Determines the TFQ score for the specified text. This score will
 * be stored internally, in the field {@link #_currentScore}.
 *
 * @param input
 *    the text to determine the TFQ score for,
 *    can be <code>null</code>.
 */
private final int computeScore(String input) {

   // TODO: This method must be made thread-safe!

   // Set starting point of the score
   _currentScore = 0xf384b2a7;

   // Loop through the string, processing one char at a time
   int length = input == null ? 0 : input.length;
   for (int i = 0; i < length; i++) {

      // Determine the next character
      int next = input.charAt(i);

      // Compute the new value
      _currentScore = _currentScore ^ next;
   }
}
```

6.7 Documentation comments

Documentation comments describe classes, interfaces and members for those who must *use* it and those who must *maintain* it. The Javadoc tool parses these comments and translates them to API documentation.

Documentation comments MUST start with:

```
/**
```

and MUST end with:

```
*/
```

Both MUST be on a separate line. The first character on the first line (the slash) MUST be properly indented, while the asterisk on the last line MUST be aligned with the first asterisk on the first line.

The lines between that first and last line MUST start with white space, followed by an asterisk "*". This asterisk MUST be aligned with the first asterisk in the first line of the documentation comment. For example:

```
/**
 * Group of members to be considered associated. None of these
 * may be considered for association with an NFG object.
 */
private final Set _associatedMembers;
```

Each class, interface and member SHOULD have an associated documentation comment, just before its declaration. This includes public, private, package-private and protected members.

For more information on Javadoc, see: http://java.sun.com/j2se/javadoc/.

6.7.1 First sentence

There MUST be at least one sentence in a documentation comment. This first sentence is used in the summaries in the generated Javadoc documentation.

This first sentence MUST NOT contain any HTML tags other than `<em>` and `<code>`. It MUST NOT contain any dots and it MUST NOT contain any special tags such as `{@link}`.

6.7.2 Javadoc tags

If applicable, `@link` tags in the description SHOULD be preferred over `@see` tags, because with `@link` tags the relation with the referenced class, interface or member can be put in context.

The text for `@deprecated` tags SHOULD indicate since which version of the module the item is considered deprecated. For an example, see section 10.3.

6.7.3 HTML tags

Documentation comments MUST NOT contain any HTML tags other than the following. All HTML tags MUST be formatted in lower case.

HTML tag	*Closed tag*	*Usage*
`<a>`	Yes	Inserts a link to a resource at a specific URL.
` `	No	Forces a line break.
`<p>`	No	Starts a new paragraph. Use this instead of multiple ` ` tags.
`<em>`	Yes	For *emphasizing* text. Use this instead of `<i>` tags.
`<strong>`	Yes	For **strong** text. Use this instead of `<b>` tags.
`<code>`	Yes	For in-line source code. Use this instead of `<tt>` tags.
`<pre>`	Yes	For multiple-line, pre-formatted source code.
`<blockquote>`	Yes	For indenting a block of text or code.
`<ul>`	Yes	For unordered lists. Each item should be marked with an `<li>` tag.
`<ol>`	Yes	For ordered lists. Each item should be marked with an `<li>` tag.
`<li>`	No	List item. Used within `<ol>` and `<ul>` elements to mark the start of an item.
`<dl>`	Yes	List of data definitions. Contains `<dt>` and `<dd>` tags.
`<dt>`	No	Type within a `<dl>` element. Should have a matching `<dd>` tag right after the type text.
`<dd>`	No	Definition for a type. Comes within a `<dl>` tag, always after the `<dt>` tag that is being described.
`<h2>`	Yes	Second-level header.
`<h3>`	Yes	Third-level header.

Header tags (`<h2>` and `<h3>`) tags SHOULD only be used in class comments, since the Javadoc tool uses header tags already, throughout the generated API documentation below the class comments.

The attribute `href` MAY be used on the `<a>` tag. The `id` tag MAY be used on all tags. Other attributes MUST NOT be used. All attribute names MUST be in lower case.

Below is an example of a documentation comment that uses some of these HTML tags:

```
/**
 * Parses the specified character string as XML. The result of the
 * parsing is stored <strong>internally</strong> and is
 * <em>not</em> returned.
 *
 * <p>The following situations cause the error flag to be set:
 *
 * <ul>
 *    <li><code>xml == null</code>
 *    <li>the string cannot be parsed
 * </ul>
 *
 * @param xml
 *    the string to be parsed, cannot be <code>null</code>.
 */
```

6.7.4 Package documentation comments

Each Java package MUST be documented in a `package.html` file, which resides in the directory for that package.

The package comment MUST describe the role and function of the package as a whole. The first sentence MUST summarize that. This first sentence MUST NOT start with a verb.

Example `package.html` file:

```
<html>
   <body>Regular expression functionality for JRuby scripts</body>
</html>
```

6.7.5 Class documentation comments

Each class comment MUST contain the following, in order:

1. The REQUIRED one-line class summary, starting with a noun.
2. An OPTIONAL continued description of the class.
3. A REQUIRED empty line.
4. A REQUIRED `@version` tag
5. At least one REQUIRED `@author` tag.
6. An OPTIONAL combination of an empty line and one or more `@see` tags

The `@version` tag must be formatted as follows:

```
@version $Revision$ $Date$
```

CVS will replace these with the number and date of the current version.

There MAY be multiple `@author` tags. Each of these tags SHOULD be in the form:

```
@author <a href="mailto:address">name</a>
```

where *`name`* is the name of the author and *`address`* is the e-mail address.

Each of the `@see` tags MUST be on a separate line.

Example of a properly formatted class documentation comment:

```
/**
 * Main class for the application. This is the second sentence,
 * which will typically <em>not</em> show up in summaries.
 *
 * <p>The main <em>visual</em> component for the application
 * is {@link MainFrame}.
 *
 * @version $Revision$ $Date$
 * @author <a href="mailto:john@doe.org">John Doe</a>
 *
 * @see Main2
 * @see javax.swing.JDesktop
 */
```

6.7.6 Field documentation comments

The documentation comment for a class or instance field MUST describe the purpose of that field and it MUST start with a noun.

The comment SHOULD describe which values the field could take, especially whether it can become `null` or not.

Unless the field is `final`, the comment SHOULD also describe the life cycle of the field, including the possible values it can take during the different stages.

Example:

```
/**
 * Name of this instance.
 *
 * <p>The value of this field changes as follows, depending
 * on the state of the object:
 *
 * <ul>
 *    <li>Initially the value is <code>null</code>.
 *    <li>After initialization (see {@link #init()} the value is
 *        not <code>null</code> and not an empty string.
 *    <li>After disposal (see {@link #deinit()}) the value is
 *        again <code>null</code>.
 */
```

6.7.7 Method documentation comments

The documentation comment for a class or instance method MUST describe the function of that method and it MUST start with a verb.

The comment MUST document:

- Each parameter with an `@param` tag, if there are any.
- The return value with an `@return` tag, if there is one.
- Each throws-clause with an `@throws` tag, if there are any.

7. Declarations

This chapter describes how declarations should be treated.

7.1 Placement

There SHOULD be only one declaration per line. For example:

```
int level;
int size;
int vector[];
```

Declarations SHOULD only be placed at the beginning of blocks, below implementation comments and precondition checks (if any). The only exceptions are `for` loops.

Example of properly formatted code:

```
public String determineLocale(String language, String region) {

   // Check preconditions
   MandatoryArgumentChecker.check("language", language);

   // Declare the variable that will contain the complete locale
   String locale;

   // If there is no region, then the locale is just the language
   if (region == null) {
      locale = language;

   // Otherwise the locale is a combination of language and region
   } else {
      locale = language + '_' + region;
   }

   return locale;
}
```

And another example, this time including a `for` loop:

```
public void computeDroneRatio(int droneBase) {

   int valueQ;
   int valueN;

   // Below the threshold, use the 'Vessian algorithm'
   if (droneBase < _ratioQ) {
      int vessian = (_ratioQ + droneBase) * getVessianBase();
      valueQ = vessian * getBaseQ();
      valueN = vessian * getBaseN();
      for (int i = 0, int last = getVessianLength(); i < last; i++) {
         valueN = valueN ^ getIndexed(i);
      }

   // Above the threshold, just multiply the base
   } else {
      valueQ = droneBase * getBaseQ();
      valueN = droneBase * getBaseN();
   }
}
```

7.2 Hiding higher-level declarations

Declarations SHOULD NOT hide higher-level declarations.

7.3 Modifiers

Declarations MUST respect the following order for modifiers:

- the access modifier – e.g. `public`, `private` or `protected`
- the `static` modifier
- the `final` modifier
- the `synchronized` modifier

Example:

```
public static final synchronized void update() {
   update(0L);
}
```

7.4 Package declarations

The package declaration is REQUIRED in every Java file. Without it, a file is in the default namespace, which is considered bad practice.

The definition MUST be right below the file header, starting on the first position of the line, and match the following form:

```
package packagename;
```

where *`packagename`* is the name of the package.

7.5 Class and interface declarations

Classes SHOULD explicitly indicate from which class they derive, even if that is class `Object`. For example:

```
public class Utils extends Object {
   ...
}
```

Anonymous inner classes SHOULD NOT be used. Instead, use named inner classes.

7.6 Constructor declarations

Each class MUST define at least one constructor.

All constructors in abstract classes MUST be marked as `protected`, to clearly indicate they are not usable except for subclasses.

7.7 Method declarations

There MUST be no space between a method name and the opening parenthesis for the parameter list. Also, the comma separating multiple parameter declarations MUST NOT immediately be followed by the next parameter. Instead, there should be white space or a new line character (in case of wrapping).

Example:

```
interface CoordinatesStore {

   /**
    * Stores the specified coordinates.
    *
    * @param x
    *    the X-coordinate.
    *
    * @param y
    *    the Y-coordinate.
    */
   void store(int x, int y);
}
```

Technically, a static initializer is just another class function with the name `<clinit>` (including the brackets). Static initializers MUST be declared above all other class functions.

7.8 Fields

Instance fields SHOULD be initialized in constructors, except if the initial value matches the initial value Java gives those fields:

- integer number (`byte`, `short`, `int` and `long`) fields are initialized to `0`;
- floating point number (`float` and `double`) fields are initialized to `0.0`;
- `boolean` fields are initialized to `false`;
- object instance fields (including arrays) are initialized to `null`.

7.9 Labels

Labels SHOULD NOT be used at all.

8. Statements

8.1 Statements per line

A line SHOULD NOT contain more than one statement.

8.2 Imports

Import statements must be in the form:

```
import packagename.classname;
```

where *packagename* is the package name for the imported class and *classname* is the unqualified name for the imported class. Together they form the so-called *fully qualified class name*.

Imports SHOULD NOT be in the following form:

```
import packagename.*;
```

Imports SHOULD be grouped per library, in alphabetical order, with an empty line separating the groups. The only exceptions are the imports from the Java platform libraries; these SHOULD be listed above all others.

Within each group of imports, classes SHOULD be sorted alphabetically.

Here is an example of proper grouping and ordering:

```
import java.util.Calendar;
import java.util.Date;
import javax.swing.JButton;
import javax.swing.JPanel;

import com.turing.State;
import com.turing.TuringMachine;

import org.apache.log4j.Logger;
import org.apache.log4j.LogLevel;
```

Unused import statements SHOULD be removed.

8.3 Compound statements

Compound statements are statements that contain one or more other statements.

Curly brackets MUST enclose the statements contained within compound statements, even if there is only one contained statement. The first curly bracket must be on the last line of the compound statement. There MUST be one space between the end of the compound statement and the curly bracket.

Example of invalid formatting:

```
if (isPainted() && isRenovated())
   cancelRenovation(ALREADY_RENOVATED); // AVOID! ERROR-PRONE!
```

Example of valid formatting:

```
if (isPainted() && isRenovated()) {
   cancelRenovation(ALREADY_RENOVATED);
```

```
}
```

Omitting braces with compound statements is a common cause for bugs.

The opening brace MUST appear at the end of the declaration statement, on the same line. Some examples with the documentation comments left out:

```
public void getWobley() {
   return _wobley;
}

public int subset(String sourceID, String targetID,
                  int selection, int index, int offset, int base) {

   // TODO: Implement
   return DEFAULT_SUBSET;
}

public void performAnalysis(DataSource source)
throws IllegalArgumentException {
   performAnalysis(source, null, null, 0L);
}
```

8.4 Return statements

Return statements MUST only be used to control flow and/or to return a value to the calling method. There MUST NOT be an empty return statement at the very end of a method.

Methods that return a value SHOULD contain only one return statement, at the end of the method. For example:

```
public String toString() {
   String description;
   if (_type == RANDOM) {
      description = “Random group”;
   } else if (_type == ORDERED) {
      description = “Ordered group”;
   } else {
      description = “Group”;
   }

   return description + “ “ + _id;
}
```

8.5 Conditional statements

Switch statements SHOULD not be used; the syntax for switch statements is complex and error-prone. Use `if` statements instead.

Conditional statements MUST always be formatted as compound statements. For example:

```
if (index < threshold) {
   adjustSetting();
}
```

If there is an `else` condition, then it MUST be on the same line as the closing bracket of the `if` statement. There MUST be one space between the closing bracket of the `if` and the `else`. For example:

```
// List not yet initialized
if (_lists == null) {
   return NEVER;

// List empty, so elements have been removed
} else if (_lists.size() == 0) {
   return NOT_ANYMORE;

// There is at least one element
} else {
   return SOME;
```

```
}
```

8.6 Loops

All loops MUST be formatted as compound statements.

8.6.1 For-loops

All `for` statements SHOULD be in the following form:

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

The *initialization* code MAY contain multiple initialization statements, using the comma character as a separator.

8.6.2 While-loops

All `while` statements SHOULD be in the following form:

```
while (condition) {
   body
}
```

8.6.3 Do-while-loops

All `do…while` statements SHOULD be in the following form:

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

8.7 Try-catch-finally statements

All `try` statements SHOULD be formatted similar to one of the following examples:

```
try {
   body
} catch (ExceptionClass exception) {
   catchblock
}

try {
   body
} finally {
   finallyblock
}

try {
   body
} catch (ExceptionClass1 exception) {
   catchblock1
} catch (ExceptionClass2 exception) {
   catchblock2
} finally {
   finallyblock
}
```

No exceptions SHOULD be thrown from a finally clause.

Example:

```
    private RoomDesign getDesign(int roomID)
    throws DataException {

       DirectConnection connection = null;

       // Fetch the room design from the server
       try {
          connection = openConnection(_connectionInfo);
          return retrieveDesign(connection, roomID);
       } catch (DirectConnectException exception) {
          throw new DataException(exception);

       // Always attempt to close the connection after usage
       } finally {
          if (connection != null) {
             try {
                connection.close();
             } catch (Throwable exception) {
                Utils.logIgnoredException(exception);
             }
          }
       }
    }
```

8.8 Casts

Casts MUST not be immediately be followed by the expression that is being cast. Example of proper formatting:

```
listener.callback((List) collection, (int) millis);
```

8.9 Operators

Unary operators (such as the unary minus, increment and decrement operators) MUST NOT be separated from their operands.

All binary operators except the dot-operator SHOULD be separated from their operands by white space.

Example of some properly formatted code:

```
total += itemExVAT + itemVAT;
count++;

if (abs < 0) {
   abs = -abs;
}
```

The order of operations SHOULD be clarified using parenthesis. For example:

```
int score = (groups[i] / quantity) + (groups[i + 1] / qauality);
```

9. Formatting

This chapter defines rules for the internal formatting of Java source files.

9.1 Character use

Both control characters and non-ASCII characters SHOULD NOT be used, except for the *new line* characters. All white space MUST be composed of spaces (ASCII character 32). Consequently, tab characters MUST NOT be used.

To include non-ASCII and/or control characters in text strings, use the Unicode escape notation, for example:

```
String text = "En Espa\u00f1ol.";
```

instead of:

```
String text = "En Español.";
```

The latter format is not recommended, since compatibility with Java compilers varies.

To determine the Unicode value of a character, see the *Unicode Character Code Charts* at http://www.unicode.org/charts/.

9.2 Indentation

Code and comments within a block SHOULD be indented in steps of 3 space characters. For example:

```
while (done == false) {
   for (int trip = (tripCount – 1); trip >= 0; i--) {
      for (int step = 0; step < threshold; j++) {
         doStep();
      }
      performedTrip();
   }
   done = check();
}
```

9.3 Line length and wrapping

Both comment and code lines SHOULD NOT extend past the limit of 78 characters. This will insure proper handling by various terminals, tools and printers.

When writing code examples, try to keep the lines even shorter, if possible. When these examples are included in documentation, they may already be some margins in place, reducing the space available to the example code.

If an expression does not fit on a single line, it MAY be considered renaming one or more of the involved entities, such as variable or method names.

In order to avoid violating the maximum line length or to improve readability of code, statements MAY be wrapped. Wrapping SHOULD comply with the following rules (in order):

1. Break after a comma.
2. Break before an operator.
3. Prefer higher-level breaks to lower-level breaks.
4. Align the new line with the beginning of the expression at the same level on the previous line.

Below is an example of wrapping of the *question-mark-colon* operator:

```
Ticket ticket = firstClassAllowed
              ? new BusinessClassTicket(ticketTemplate)
              : new EconomyClassTicket(ticketTemplate);
```

Here are 2 more examples, showing the wrapping of method calls:

```
function(longExpression, veryLongExpression, extremelyLongExpression,
         anotherParameter, yetAnotherParameter);

long result = compute(longExpression,
                      function(extremelyLongExpression1,
                               extremelyLongExpression2));
```

Here is an example of wrapping and indentation in case of a long method declaration. Notice that the first parameter is already wrapped:

```
protected final synchronized AutomaticUpdateProcess constructUpdateProcess(
   String id,           String   processName,
   String programName,  String   resultDescription,
   String extraContent, MetaData metaData)

throws IllegalArgumentException,
       IllegalStateException,
       ProcessCreationException {

   constructUpdateProcess(id, processName, programName, resultDescription,
                          extraContent, metaData, null);
```

9.4 White space

White space (both vertical and horizontal) SHOULD be used to make code more readable.

Lines MUST NOT end with white space characters.

Individual method declarations MUST be separated by one blank line. For example:

```
/**
 * Resets the current average.
 */
public void resetAverage() {
   _averageCount = 0;
}

/**
 * Resets the current index.
 */
public void resetIndex() {
   _index = 0;
}
```

9.5 Various maximums

Large files and large methods are signals that indicate the code is probably hard to understand and modularity is probably poor. Knowing that, some maximums are suggested:

- The length of a Java source file SHOULD NOT exceed 2000 lines.
- The length of the content of a method SHOULD NOT exceed 30 lines.
- The number of parameters for a method SHOULD NOT exceed 6.

10. Examples

This chapter gives some examples of properly formatted source files.

10.1 An interface

Below is an example of a properly formatted Java source file defining an interface.

```
/*
 * $Id$
 *
 * Copyright 2003-2006 Orange Nederland Breedband B.V.
 * See the COPYRIGHT file for redistribution and use restrictions.
 */
package nl.orange.bb.common;

import java.util.EventListener;

/**
 * Listener for back-office security events.
 *
 * @version $Revision$ $Date$
 * @author <a href="john.doe@orange-ft.com">John Doe</a>
 *
 * @see MainFrame
 */
public interface SecurityEventListener extends EventListener {

   /**
    * Requests this listener to call back the initiator of the
    * security event.
    *
    * @param event
    *    the {@link SecurityEvent} that specifies which service to
    *    call back, should never be <code>null</code>.
    */
   void callback(SecurityEvent event);
}
```

10.2 Simple class

Below is an example of a properly formatted Java source file containing just one simple class.

```
/*
 * $Id$
 *
 * Copyright 2003-2006 Orange Nederland Breedband B.V.
 * See the COPYRIGHT file for redistribution and use restrictions.
 */
package org.xins.common;

/**
 * Exception that indicates the total time-out for a service call was reached.
 *
 * @version $Revision: 1.6 $ $Date: 2006/03/21 14:33:59 $
 * @author <a href="mailto:jane.doe@xins.org">Jane Doe</a>
 *
 * @since XINS 1.0.0
 */
public final class TimeOutException extends Exception {


   //-------------------------------------------------------------------------
```

```
   // Constructors
   //-----------------------------------------------------------------------

   /**
    * Constructs a new <code>TimeOutException</code>.
    */
   public TimeOutException() {
      // empty
   }
}
```

10.3 Complex class

Below is an example of a properly formatted Java source file containing a fairly complex class, with an inner class. This is a slightly modified version of an actual class which is part of XINS.

```
/*
 * $Id$
 *
 * Copyright 2003-2006 Orange Nederland Breedband B.V.
 * See the COPYRIGHT file for redistribution and use restrictions.
 */
package org.xins.client;

import java.util.ArrayList;
import java.util.Collections;
import java.util.Iterator;
import java.util.List;

import org.xins.common.MandatoryArgumentChecker;
import org.xins.common.Utils;
import org.xins.common.text.TextUtils;
import org.xins.common.xml.Element;
import org.xins.common.xml.ElementBuilder;

/**
 * Element in a XINS result data section.
 *
 * <p>Note that this class is not thread-safe. It should not be used from
 * different threads at the same time. This applies even to read operations.
 *
 * <p>Note that the namespace URIs and local names are not checked for
 * validity in this class.
 *
 * @version $Revision$ $Date$
 * @author <a href="mailto:tom@jerry.tv">Tom Jerry</a>
 * @author <a href="mailto:jerry.thomas@aol.com">Jerry Thomas</a>
 *
 * @since XINS 1.0.0
 */
public class DataElement implements Cloneable {

   // NOTE: The behavior of this class has been slightly redefined in XINS
   //       1.1. In XINS 1.0, the name for a DataElement was a combination of
   //       the namespace prefix and the local name. In XINS 1.1, the name is
   //       just the local name. Since XINS 1.0 did not support XML Namespaces
   //       yet, this is not considered an incompatibility.

   // TODO: Check the validity of the 'local name' since it can currently be
   //       anything, e.g. an empty string "" or some text including spaces,
   //       such as " bla dee bla"

   //-----------------------------------------------------------------------
   // Class fields
   //-----------------------------------------------------------------------

   /**
    * Fully-qualified name of this class. Never <code>null</code>.
    */
   private static final String CLASSNAME = DataElement.class.getName();
```

```
//---------------------------------------------------------------------------
// Class methods
//---------------------------------------------------------------------------

/**
 * Creates a new <code>DataElement</code> with no namespace and the
 * specified local name.
 *
 * @param localName
 *    the local name of the element, cannot be <code>null</code>.
 *
 * @return
 *    a new <code>DataElement</code> instance that matches the specified
 *    local name and has no name space, never <code>null</code>.
 *
 * @throws IllegalArgumentException
 *    if <code>localName == null</code>.
 */
public static DataElement createDataElement(String localName)
throws IllegalArgumentException {

   // NOTE: Argument checking is done by the constructor

   return new DataElement(null, localName);
}


//---------------------------------------------------------------------------
// Constructors
//---------------------------------------------------------------------------

/**
 * Creates a new <code>DataElement</code>.
 *
 * @param namespaceURI
 *    the namespace URI for the element, can be <code>null</code>; an empty
 *    string is equivalent to <code>null</code>.
 *
 * @param localName
 *    the local name of the element, cannot be <code>null</code>.
 *
 * @throws IllegalArgumentException
 *    if <code>localName == null</code>.
 */
DataElement(String namespaceURI, String localName)
throws IllegalArgumentException {

   // Check preconditions
   MandatoryArgumentChecker.check("localName", localName);

   // An empty namespace URI is equivalent to null
   if (namespaceURI != null && namespaceURI.length() == 0) {
      namespaceURI = null;
   }

   // Store namespace URI and local name
   _namespaceURI = namespaceURI;
   _localName    = localName;
}


//---------------------------------------------------------------------------
// Fields
//---------------------------------------------------------------------------

/**
 * URI of the namespace. This field can be <code>null</code>, but it can
 * never be an empty string.
 */
private final String _namespaceURI;

/**
 * Local name. This field is never <code>null</code>, but it can be an
 * empty string.
 */
```

```
private final String _localName;

/**
 * List of child elements. This field is lazily initialized and is
 * initially <code>null</code>.
 */
private ArrayList _children;

/**
 * Character content for this element. Initially <code>null</code>.
 */
private String _text;


//--------------------------------------------------------------------------
// Methods
//--------------------------------------------------------------------------

/**
 * Gets the namespace URI.
 *
 * @return
 *    the namespace URI for this element, or <code>null</code> if there is
 *    none, but never an empty string.
 *
 * @since XINS 1.1.0
 */
public String getNamespaceURI() {
   return _namespaceURI;
}

/**
 * Gets the local name.
 *
 * @return
 *    the local name of this element, cannot be <code>null</code>.
 *
 * @since XINS 1.1.0
 */
public String getLocalName() {
   return _localName;
}

/**
 * Gets the local name.
 *
 * @return
 *    the local name of this element, cannot be <code>null</code>.
 *
 * @deprecated
 *    Deprecated since XINS 1.1.0. Use {@link #getLocalName()} instead,
 *    which has the same functionality and behavior. This method has been
 *    deprecated since it returned a combination of the namespace prefix
 *    and the local name in XINS 1.0. This method is guaranteed not to be
 *    removed before XINS 2.0.0.
 */
public String getName() {
   return getLocalName();
}

/**
 * Adds a new child element.
 *
 * @param child
 *    the new child to add to this element, cannot be <code>null</code>.
 *
 * @throws IllegalArgumentException
 *    if <code>child == null || child == <em>this</em></code>.
 */
void addChild(DataElement child) throws IllegalArgumentException {

   // Check preconditions
   MandatoryArgumentChecker.check("child", child);
   if (child == this) {
      String thisMethod    = "addChild(DataElement)";
```

```
      String subjectClass  = Utils.getCallingClass();
      String subjectMethod = Utils.getCallingMethod();
      String detail        = "child == this";
      Utils.logProgrammingError(CLASSNAME,    thisMethod,
                                subjectClass, subjectMethod,
                                detail);
      throw new IllegalArgumentException(detail);
   }

   // Lazily initialize
   if (_children == null) {
      _children = new ArrayList();
   }

   // Add the child to the list
   _children.add(child);
}

/**
 * Gets the list of all child elements.
 *
 * @return
 *    an unmodifiable {@link List} containing all child elements; each
 *    element in the list is another <code>DataElement</code> instance;
 *    never <code>null</code>.
 */
public List getChildElements() {

   List children;

   // If there are no children, then return an immutable empty List
   if (_children == null || _children.size() == 0) {
      children = Collections.EMPTY_LIST;

   // Otherwise return an immutable view of the list of children
   } else {
      children = Collections.unmodifiableList(_children);
   }

   return children;
}

/**
 * Gets the list of child elements that match the specified name.
 *
 * @param name
 *    the name for the child elements to match, cannot be
 *    <code>null</code>.
 *
 * @return
 *    a {@link List} containing each child element that matches the
 *    specified name as another <code>DataElement</code> instance;
 *    never <code>null</code>.
 *
 * @throws IllegalArgumentException
 *    if <code>name == null</code>.
 */
public List getChildElements(String name)
throws IllegalArgumentException {

   // XXX: Namespaces are not supported

   // Check preconditions
   MandatoryArgumentChecker.check("name", name);

   List matches;

   // If there are no children, then return an empty List instance
   if (_children.size() == 0) {
      matches = Collections.EMPTY_LIST;

   // There are children, find all matching ones
   } else {
      matches = new ArrayList();
      Iterator it = _children.iterator();
```

```
      while (it.hasNext()) {
         DataElement child = (DataElement) it.next();
         if (name.equals(child.getLocalName())) {
            matches.add(child);
         }
      }

      // If there are no matching children, return an empty List instance
      if (matches.size() == 0) {
         matches = Collections.EMPTY_LIST;

      // Otherwise return an immutable list with all matches
      } else {
         matches = Collections.unmodifiableList(matches);
      }
   }

   return matches;
}

/**
 * Sets the character content. The existing character content, if any, is
 * replaced
 *
 * @param text
 *    the character content for this element, or <code>null</code> if there
 *    if none.
 */
void setText(String text) {
   _text = text;
}

/**
 * Gets the character content, if any.
 *
 * @return
 *    the character content of this element, or <code>null</code> if no
 *    text has been specified for this element.
 */
public String getText() {
   return _text;
}

/**
 * Converts this DataElement to an {@link org.xins.common.xml.Element}
 * object.
 *
 * @return
 *    the converted object, never <code>null</code>.
 *
 * @since XINS 1.3.0
 */
public Element toXMLElement() {
   return toXMLElement(this);
}

/**
 * Converts the given DataElement to a
 * {@link org.xins.common.xml.Element} object.
 *
 * @param dataElement
 *    the input element to convert, cannot be <code>null</code>
 *
 * @return
 *    the converted object, never <code>null</code>.
 *
 * @throws IllegalArgumentException
 *    if <code>dataElement == null</code>.
 */
private Element toXMLElement(DataElement dataElement)
throws IllegalArgumentException {

   // Check preconditions
   MandatoryArgumentChecker.check("dataElement", dataElement);
```

```
   String elementName = dataElement.getLocalName();
   String elementNameSpaceURI = dataElement.getNamespaceURI();
   String elementText = dataElement.getText();
   List elementChildren = dataElement.getChildElements();

   ElementBuilder builder = new ElementBuilder(elementNameSpaceURI, elementName);

   builder.setText(elementText);

   // Add the children of this element
   Iterator itChildren = elementChildren.iterator();
   while (itChildren.hasNext()) {
      DataElement nextChild = (DataElement) itChildren.next();
      Element transformedChild = toXMLElement(nextChild);
      builder.addChild(transformedChild);
   }

   return builder.createElement();
}

/**
 * Clones this object. The clone will have the same namespace URI and local
 * name and equivalent children and character content.
 *
 * @return
 *    a new clone of this object, never <code>null</code>.
 */
public Object clone() {

   // Construct a new DataElement, copy all field values (shallow copy)
   DataElement clone;
   try {
      clone = (DataElement) super.clone();
   } catch (CloneNotSupportedException exception) {
      String detail = null;
      throw Utils.logProgrammingError(DataElement.class.getName(),
                                      "clone()",
                                      "java.lang.Object",
                                      "clone()",
                                      detail,
                                      exception);
   }

   // Deep copy the children
   if (_children != null) {
      clone._children = (ArrayList) _children.clone();
   }

   return clone;
}


//---------------------------------------------------------------------------
// Inner classes
//---------------------------------------------------------------------------

/**
 * Qualified name for an element or attribute. This is a combination of an
 * optional namespace URI and a mandatory local name.
 *
 * @author <a href="mailto:jerry.thomas@aol.com">Jerry Thomas</a>
 *
 * @since XINS 1.1.0
 */
public static final class QualifiedName extends Object {

   //-------------------------------------------------------------------------
   // Constructors
   //-------------------------------------------------------------------------

   /**
    * Constructs a new <code>QualifiedName</code> with the specified
    * namespace and local name.
    *
    * @param namespaceURI
```

```
 *    the namespace URI for the element, can be <code>null</code>; an
 *    empty string is equivalent to <code>null</code>.
 *
 * @param localName
 *    the local name of the element, cannot be <code>null</code>.
 *
 * @throws IllegalArgumentException
 *    if <code>localName == null</code>.
 */
public QualifiedName(String namespaceURI, String localName)
throws IllegalArgumentException {

   // Check preconditions
   MandatoryArgumentChecker.check("localName", localName);

   // An empty namespace URI is equivalent to null
   if (namespaceURI != null && namespaceURI.length() < 1) {
      namespaceURI = null;
   }

   // Initialize fields
   _hashCode     = localName.hashCode();
   _namespaceURI = namespaceURI;
   _localName    = localName;
}


//--------------------------------------------------------------------------
// Fields
//--------------------------------------------------------------------------

/**
 * Hash code for this object.
 */
private final int _hashCode;

/**
 * Namespace URI. Can be <code>null</code>.
 */
private final String _namespaceURI;

/**
 * Local name. Cannot be <code>null</code>.
 */
private final String _localName;


//--------------------------------------------------------------------------
// Methods
//--------------------------------------------------------------------------

/**
 * Returns the hash code value for this object.
 *
 * @return
 *    the hash code value.
 */
public int hashCode() {
   return _hashCode;
}

/**
 * Compares this object with the specified object for equality.
 *
 * @param obj
 *    the object to compare with, or <code>null</code>.
 *
 * @return
 *    <code>true</code> if this object and the argument are considered
 *    equal, <code>false</code> otherwise.
 */
public boolean equals(Object obj) {

   boolean equal = false;
```

```
            // To be equal, the object must be an instance of the same class...
            if (obj instanceof QualifiedName) {

               QualifiedName qn = (QualifiedName) obj;

               // ...and the local name must be the same...
               if (_localName.equals(qn._localName)) {

                  // ...and the name spaces must match.
                  if (_namespaceURI == null) {
                     equal = qn._namespaceURI == null;
                  } else {
                     equal = _namespaceURI.equals(qn._namespaceURI);
                  }
               }
            }

            return equal;
         }

         /**
          * Gets the namespace URI.
          *
          * @return
          *    the namespace URI, can be <code>null</code>.
          */
         public String getNamespaceURI() {
            return _namespaceURI;
         }

         /**
          * Gets the local name.
          *
          * @return
          *    the local name, never <code>null</code>.
          */
         public String getLocalName() {
            return _localName;
         }
      }
   }
```