**Java Coding Guidelines**

|  |
| --- |
|  |
|  |

**Table of Contents**

[1. Introduction 5](#_Toc386441633)

[1.1. Why Coding Standards are Important 5](#_Toc386441634)

[1.2. When you go against a standard, document it 5](#_Toc386441635)

[2. File Names 5](#_Toc386441636)

[2.1. File SUFFIXES 5](#_Toc386441637)

[2.2. Common File Names 6](#_Toc386441638)

[3. File Organization 6](#_Toc386441639)

[3.1. Java Source Files 6](#_Toc386441640)

[3.1.1. Beginning Comments 6](#_Toc386441641)

[3.1.2. Package and Import Statement 8](#_Toc386441642)

[3.1.3. Class and Interface Declarations 9](#_Toc386441643)

[4. Indentation 9](#_Toc386441644)

[4.1. Line Lengths 9](#_Toc386441645)

[4.2. Wrapping Lines 10](#_Toc386441646)

[5. Comments 13](#_Toc386441647)

[5.1. Implementation Comments Format 13](#_Toc386441648)

[5.2. Block Comments 13](#_Toc386441649)

[5.3. Single Line Comments 14](#_Toc386441650)

[5.4. Trailing Comments 15](#_Toc386441651)

[5.5. End of Line Comments 15](#_Toc386441652)

[5.6. Documentation Comments 16](#_Toc386441653)

[6. Declarations 18](#_Toc386441654)

[6.1. Number per Line 18](#_Toc386441655)

[6.2. Initialization 18](#_Toc386441656)

[6.3. Placement 19](#_Toc386441657)

[6.4. Class and Interface Declarations 20](#_Toc386441658)

[7. Statements 21](#_Toc386441659)

[7.1. Simple Statements 21](#_Toc386441660)

[7.2. Compound Statements 21](#_Toc386441661)

[7.3. Return Statements 21](#_Toc386441662)

[7.4. For Statements 22](#_Toc386441663)

[7.5. While Statements 23](#_Toc386441664)

[7.6. Do\_While Statements 23](#_Toc386441665)

[7.7. Switch Statements 24](#_Toc386441666)

[7.8. Try-Catch Statements 25](#_Toc386441667)

[8. White Space 26](#_Toc386441668)

[8.1. Blank Lines 26](#_Toc386441669)

[8.2. Blank Spaces 26](#_Toc386441670)

[9. Naming Conventions 27](#_Toc386441671)

[9.1. Jfc Naming Conventions 30](#_Toc386441672)

[10. Member Function Visibility 32](#_Toc386441673)

[10.1. Field Visibility 33](#_Toc386441674)

[11. Standards for Classes 34](#_Toc386441675)

[11.1. Class Visibility 34](#_Toc386441676)

[11.2. Use package visibility for classes internal to a component. 34](#_Toc386441677)

[11.3. Use public visibility for the facades of components. 34](#_Toc386441678)

[11.4. Apply the “final” keyword sensibly 34](#_Toc386441679)

[11.5. Use Wild Cards When Importing Classes 34](#_Toc386441680)

[11.6. Alternative – Explicitly Specify Each Imported Class 34](#_Toc386441681)

[12. Servlets 35](#_Toc386441682)

[13. Code Conventions for JavaServer Pages 36](#_Toc386441683)

[13.1. File Name and Locations 36](#_Toc386441684)

[13.2. File Organization 37](#_Toc386441685)

[13.3. Opening Comments 38](#_Toc386441686)

[13.4. JSP Page Directive(s) 39](#_Toc386441687)

[13.5. Optional Tag Library Directive (s) 41](#_Toc386441688)

[13.6. Optional JSP Declaration (S) 42](#_Toc386441689)

[13.7. HTML and JSP code 42](#_Toc386441690)

[13.8. Comments 47](#_Toc386441691)

[13.9. JSP declarations 49](#_Toc386441692)

[13.10. JSP SCriptlets 50](#_Toc386441693)

[13.11. JSP expressions 53](#_Toc386441694)

[13.12. White Space 54](#_Toc386441695)

[13.13. Blank Lines 54](#_Toc386441696)

[13.14. Blank Spaces 55](#_Toc386441697)

[13.15. Naming Conventions 56](#_Toc386441698)

[13.16. JSP Names 56](#_Toc386441699)

[13.17. TAG names 57](#_Toc386441700)

[13.18. TAG prefix names 57](#_Toc386441701)

[13.19. JSP pages in XML SYNTAX 58](#_Toc386441702)

[13.20. JSP document structure 58](#_Toc386441703)

[13.21. XML Comments 59](#_Toc386441704)

[13.22. Java code in JSP documents 59](#_Toc386441705)

[13.23. Programming Practice 60](#_Toc386441706)

[13.24. JAVA Beans initialization 60](#_Toc386441707)

[13.25. JSP Implicit TAGs 61](#_Toc386441708)

[13.26. Using Custom Tags 63](#_Toc386441709)

# Introduction

The goal of this document is to lay down the Java coding rules and guidelines that all developers at KPIT should follow to enable consistent, easily readable and maintainable code. It also strives to cultivate good coding practices among the developers. The document also covers the JSP Coding Standards. The Java Coding Standards and Guidelines laid down in this document are to be followed only in the event that the customer does not specify other coding standards.

# Why Coding Standards are Important

Coding standards for Java are important because they lead to greater consistency within your code and the code of your teammates. Greater consistency leads to code that is easier to understand, which in turn means it is easier to develop and to maintain. This reduces the overall cost of the applications that you create.

You have to remember that your Java code will exist for a long time, long after you have moved on to other projects. An important goal during development is to ensure that you can transition your work to another developer, or to another team of developers, so that they can continue to maintain and enhance your work without having to invest an unreasonable effort to understand your code. Code that is difficult to understand runs the risk of being scrapped and rewritten. If everyone is doing their own thing then it makes it very difficult to share code between developers, raising the cost of development and maintenance.

# When you go against a standard, document it

All standards, except for this one, can be broken. If you do so, you must document why you broke the standard, the potential implications of breaking the standard, and any conditions that may/must occur before the standard can be applied to this situation.

# File Names

This section will discuss the commonly used suffixes and names.

# File SUFFIXES

|  |  |
| --- | --- |
| **File Type** | **Suffix** |
| Java Source | .java |
| Java Bytecode | .class |

# Common File Names

Frequently used file names include

|  |  |
| --- | --- |
| **File Name** | **Use** |
| GNUmakefile | The preferred name for makefiles. Use gnumake to build the software |
| README | The preferred name for the file that summarizes the contents of the particular directories. |

# File Organization

A file consists of sections that should be separated by blank lines and an optional comment identifying each section. Files longer than 2000 lines are cumbersome and should be avoided.

# Java Source Files

Each Java source file contains a single public class or interface. When private classes and interfaces are associated with a public class, you can put them in the same source file as the public class. The public class should be the first class or interface in the file. Java source files have the following ordering:

* Beginning Comments
* Package and Import statements
* Class and Interface declarations

# Beginning Comments

All source files should begin with a c-style comment that lists the class name, version information, date, and copyright notice:

/ \*

\* Class Name(s) <Class Name Used in the java files>

\*

\* Configuration Identifier < the unique identifier for the product >

\*

\* Version Number <version of the product>

\*

\* Created By / Date <Name of the author(s) of the class> <DD MMM, YYYY. e.g. 06

\* Dec, 2002>

\*

\* Modification Description <The description for the modification>

\*

\* Modified By / Date <Name of the modifier(s) of the class> <Date of Modification in

\* DD MMM, YYYY format e.g. 10 Mar, 2002 >

\*/

All the class definition should start with a block comment of the format as shown below. The tag that is provided below for the class and method are mandatory and all the other tags are optional and are based on the discretion of the author.

/ \*\*

\* <Brief description of the class>

\*

\* @author <Author name here>

\*

\* @version <version of the project>

\*

\* @Date <date of creation of the class>

\*

\* <Add additional tags if required.>

\* /

All the method of the class should have the following mandatory comments

/\*\*

\* <Brief description of the method >

\*

\* @param <variable name> <description>

\*

\* @param <variable name> <description>

\*

\* @return <variable name> <description> (If return type exist)

**\*/**

# Package and Import Statement

The first non-comment line of most Java source files is a package statement. After that, import statements can follow.

Example:

package java.awt;

import java.awt.peer.CanvasPeer;

# Class and Interface Declarations

The following table describes the parts of a class or interface declaration, in the order that they should appear.

|  |  |
| --- | --- |
| **Part of Class / Interface Declarations** | **Notes** |
| Class / Interface Documentation |  |
| Class / Interface Statements |  |
| Class / Interface Implementation | This comment should contain any class wide or comment (/\*….\*/), if necessary, interface-wide information that was not appropriate |
| Class (static) Variables | First the public class variables, then the protected, then the package level (no access modifier) and then the private variables. |
| Constructors |  |
| Methods | These methods should be grouped by functionality rather than by scope or accessibility. For example, a private class method can be in-between two public instance methods. The goal is to make coding and understanding easier. |

# Indentation

Four spaces should be used as the unit of indentation. The exact construction of the indentation (Spaces vs. tabs) is unspecified. Tabs must be set exactly every 8 spaces (not 4).

# Line Lengths

Avoid lines longer than 80 characters, since they’re not handled well by many terminals and tools.

**Note:** Examples for use in documentation should have a shorter line length —generally not more than 80 characters.

# Wrapping Lines

When an expression will not fit on a single line, break it according to these general principles:

* Break after a comma
* Break before an operator
* Prefer higher-level breaks to lower-level breaks
* Align the new line with the beginning of the expression at the same level on the previous line
* If the above rules lead to confusing code or to code that’s squished up against the right margin, just indent 8 spaces instead

Here are some examples of breaking method calls:

someMethod(longExpression1, longExpression2, longExpression3,

longExpression4, longExpression5);

var = someMethod1(longExpression1,

someMethod2(longExpression2,

longExpression3));

Following are two examples of breaking an arithmetic expression. The first is preferred, since the break occurs outside the parenthesized expression, which is at a higher level.

longName1 = longName2 \* (longName3 + longName4 - longName5)

+ 4 \* longname6; // PREFER

longName1 = longName2 \* (longName3 + longName4

- longName5) + 4 \* longname6; // AVOID

Following are two examples of indenting method declarations. The first is the conventional case. The second would shift the second and third lines to the far right if it used conventional indentation, so instead it indents only 8 spaces.

//CONVENTIONAL INDENTATION

someMe

thod(int anArg, Object anotherArg, String yetAnotherArg,

Object andStillAnother) {

...

}

// INDENT 8 SPACES TO AVOID VERY DEEP INDENTS

private static synchronized horkingLongMethodName(int anArg,

Object anotherArg, String yetAnotherArg,

Object andStillAnother) {

...

}

Line wrapping for if statements should generally use the 8-space rule, since conventional (4 space) indentation makes seeing the body difficult. For example:

//DON’T USE THIS INDENTATION

if ((condition1 && condition2)

|| (condition3 && condition4)

||!(condition5 && condition6)) { // BAD WRAPS

doSomethingAboutIt(); // MAKE THIS LINE EASY TO MISS

}

// USE THIS INDENTATION INSTEAD

if ((condition1 && condition2)

|| (condition3 && condition4)

||!(condition5 && condition6)) {

doSomethingAboutIt();

}

//OR USE THIS

if ((condition1 && condition2) || (condition3 && condition4)

||!(condition5 && condition6)) {

doSomethingAboutIt();

}

Here are three acceptable ways to format ternary expressions:

alpha = (aLongBooleanExpression) ? beta : gamma;

alpha = (aLongBooleanExpression) ? beta

: gamma;

alpha = (aLongBooleanExpression)

? beta

: gamma;

# Comments

Java programs can have two kinds of comments: implementation comments and documentation comments. Implementation comments are those found in C++, which are delimited by /\*...\*/, and //. Documentation comments (known as “doc comments”) are Java-only, and are delimited by /\*\*...\*/. Doc comments can be extracted to HTML files using the javadoc tool.

Implementation comments are means for commenting out code or for comments about the particular implementation. Doc comments are meant to describe the specification of the code, from an implementation-free perspective to be read by developers who might not necessarily have the source code at hand. Comments should be used to give overviews of code and provide additional information that is not readily available in the code itself. Comments should contain only information that is relevant to reading and understanding the program. For example, information about how the corresponding package is built or in what directory it resides should not

be included as a comment.

Discussion of nontrivial or no obvious design decisions is appropriate, but avoid duplicating information that is present in (and clear from) the code. It is too easy for redundant comments to get out of date. In general, avoid any comments that are likely to get out of date as the code evolves.

**Note:** The frequency of comments sometimes reflects poor quality of code. When you feel compelled to add a comment, consider rewriting the code to make it clearer. Comments should not be enclosed in large boxes drawn with asterisks or other characters. Comments should never include special characters such as form-feed and backspace.

# Implementation Comments Format

Programs can have four styles of implementation comments: block, single-line, trailing and end of- line.

# Block Comments

Block comments are used to provide descriptions of files, methods, data structures and algorithms. Block comments may be used at the beginning of each file and before each method. They can also be used in other places, such as within methods. Block comments inside a function or method should be indented to the same level as the code they describe. A blank line to set it apart from the rest of the code should precede a block comment.

/\*

\* Here is a block comment.

\*/

Block comments can start with /\*-, which is recognized by **indent**(1) as the beginning of a block comment that should not be reformatted. Example:

/\*-

\* Here is a block comment with some very special

\* formatting that I want indent(1) to ignore.

\*

\* one

\* two

\* three

\*/

**Note:**

If you don’t use **indent**(1), you don’t have to use /\*- in your code or make any other concessions to the possibility that someone else might run **indent**(1) on your code.

# Single Line Comments

Short comments can appear on a single line indented to the level of the code that follows. If a comment can’t be written in a single line, it should follow the block comment format (see section 5.1.1). A blank line should precede a single-line comment. Here’s an example of a single-line comment in Java code:

if (condition) {

/\* Handle the condition. \*/

...

}

# Trailing Comments

Very short comments can appear on the same line as the code they describe, but should be shifted far enough to separate them from the statements. If more than one short comment appears in a chunk of code, they should all be indented to the same tab setting.

Here’s an example of a trailing comment in Java code:

if (a == 2) {

return TRUE; /\* special case \*/

} else {

return isPrime(a); /\* works only for odd a \*/

}

# End of Line Comments

The // comment delimiter can comment out a complete line or only a partial line. It shouldn’t be used on consecutive multiple lines for text comments; however, it can be used in consecutive multiple lines for commenting out sections of code. Examples of all three styles follow:

if (foo > 1) {

// Do a double-flip.

...

} else {

return false; // Explain why here.

}

//if (bar > 1) {

//

// // Do a triple-flip.

// ...

//}

//else{

// return false;

//}

# Documentation Comments

For further details, see “How to Write Doc Comments for Javadoc” which includes information on the doc comment tags (@return, @param, @see): Refer to the attached document.



Doc comments describe Java classes, interfaces, constructors, methods, and fields. Each doc comment is set inside the comment delimiters /\*\*...\*/, with one comment per class, interface, or member. This comment should appear just before the declaration:

/\*\*

\* The Example class provides ...

\*/

public class Example { ...

Notice that top-level classes and interfaces are not indented, while their members are. The first line of doc comment (/\*\*) for classes and interfaces is not indented; subsequent doc comment lines each have 1 space of indentation (to vertically align the asterisks). Members, including constructors, have 4 spaces for the first doc comment line and 5 spaces thereafter. If you need to give information about a class, interface, variable, or method that isn’t appropriate for documentation, use an implementation block comment (see section 5.1.1) or single-line (see section 5.1.2) comment immediately *after* the declaration. For example, details about the implementation of a class should go in in such an implementation block comment *following* the class statement, not in the class doc comment.

Doc comments should not be positioned inside a method or constructor definition block, because Java associates documentation comments with the first declaration *after* the comment.

# Declarations

# Number per Line

One declaration per line is recommended since it encourages commenting. In other words,

int level; // indentation level

int size; // size of table

is preferred over

int level, size;

Do not put different types on the same line.

Example:

int foo, fooarray[]; //WRONG!

**Note:** The examples above use one space between the type and the identifier. Another acceptable alternative is to use tabs, e.g.:

int level; // indentation level

int size; // size of table

Object currentEntry; // currently selected table entry

# Initialization

Try to initialize local variables where they’re declared. The only reason not to initialize a variable where it’s declared is if the initial value depends on some computation occurring first.

# Placement

Put declarations only at the beginning of blocks. (A block is any code surrounded by curly braces “{” and “}”.) Don’t wait to declare variables until their first use; it can confuse the unwary programmer and hamper code portability within the scope.

void myMethod() {

int int1 = 0; // beginning of method block

if (condition) {

int int2 = 0; // beginning of "if" block

. ..

}

}

The one exception to the rule is indexes of for loops, which in Java can be declared in the for statement:

for (int i = 0; i < maxLoops; i++) { ... }

Avoid local declarations that hide declarations at higher levels. For example, do not declare the same variable name in an inner block:

int count;

...

myMethod() {

if (condition) {

int count; // AVOID!

...

}

...

}

# Class and Interface Declarations

When coding Java classes and interfaces, the following formatting rules should be followed:

* No space between a method name and the parenthesis “(“ starting its parameter list
* Open brace “{” appears at the end of the same line as the declaration statement
* Closing brace “}” starts a line by itself indented to match its corresponding opening statement, except when it is a null statement the “}” should appear immediately after the “{“
* Methods are separated by a blank line

class Sample extends Object {

int ivar1;

int ivar2;

Sample (int i, int j) {

ivar1 = i;

ivar2 = j;

}

int emptyMethod() {}

...

}

# Statements

# Simple Statements

Each line should contain at most one statement. Example:

argv++; // Correct

argc++; // Correct

argv++; argc--; // AVOID!

# Compound Statements

Compound statements are statements that contain lists of statements enclosed in braces “{statements}”. See the following sections for examples.

* The enclosed statements should be indented one more level than the compound statement
* The opening brace should be at the end of the line that begins the compound statement; the closing brace should begin a line and be indented to the beginning of the compound statement
* Braces are used around all statements, even single statements, when they are part of a control structure, such as an if-else or for statement. This makes it easier to add statements without accidentally introducing bugs due to forgetting to add braces

# Return Statements

A return statement with a value should not use parentheses unless they make the return value more obvious in some way. Example:

return;

return myDisk.size();

return (size ? size : defaultSize);

## if, if-else, if else-if else Statements

The if-else class of statements should have the following form:

if (condition) {

statements;

}

if (condition) {

statements;

} else {

statements;

}

if (condition) {

statements;

} else if (condition) {

statements;

} else {

statements;

}

**Note:** if statements always use braces {}. Avoid the following error-prone form:

if ( condition) //AVOID! THIS OMITS THE BRACES {}!

statement;

# For Statements

A for statement should have the following form:

for (initialization; condition; update) {

statements;

}

An empty for statement (one in which all the work is done in the initialization, condition, and update clauses) should have the following form:

for (initialization; condition; update);

When using the comma operator in the initialization or update clause of a for statement, avoid the complexity of using more than three variables. If needed, use separate statements before the for loop (for the initialization clause) or at the end of the loop (for the update clause).

# While Statements

A while statement should have the following form:

while (condition) {

statements;

}

An empty while statement should have the following form:

while (condition);

# Do\_While Statements

A do-while statement should have the following form:

do {

statements;

} while (condition);

# Switch Statements

A switch statement should have the following form:

switch (condition) {

case ABC:

statements;

/\* falls through \*/

case DEF:

statements;

break;

case XYZ:

statements;

break;

default:

statements;

break;

}

Every time a case falls through (doesn’t include a break statement), add a comment where the break statement would normally be. This is shown in the preceding code example with the /\* falls through \*/ comment.

Every switch statement should include a default case. The break in the default case is redundant, but it prevents a fall-through error if later another case is added.

# Try-Catch Statements

A try-catch statement should have the following format:

try {

statements;

} catch(ExceptionClass e) {

statements;

}

A try-catch statement may also be followed by finally, which executes regardless of whether or not the try block has completed successfully.

try {

statements;

} catch(ExceptionClass e) {

statements;

} finally {

statements;

}

# White Space

# Blank Lines

Blank lines improve readability by setting off sections of code that are logically related. Two blank lines should always be used in the following circumstances:

* Between sections of a source file
* Between class and interface definitions

One blank line should always be used in the following circumstances:

* Between methods
* Between logical sections inside a method to improve readability

# Blank Spaces

A keyword followed by a parenthesis should be separated by a space. Example:

while (true) {

...

}

Note that a blank space should not be used between a method name and its opening parenthesis.

# Naming Conventions

Naming conventions make programs more understandable by making them easier to read. They can also give information about the function of the identifier—for example, whether it’s a constant, package, or class—which can be helpful in understanding the code.

|  |  |  |
| --- | --- | --- |
| **Identifier Type** | **Rules for Naming** | **Examples** |
| Packages | The prefix of the unique package name is always written in all – lower case ASCII letters and should be one of the top-level domain names. Currently com, edu,  gov, mil, net, org, or one of the  English two-letter codes identifying countries as specified in ISO Standard  3166, 1981. Subsequent components of the package name vary according to an organization’s own internal naming conventions. Such conventions might specify that certain directory name components be division, department, project, machine, or login names. |  |
| Classes | Classes name should be nouns, with mixed case with the first letter of each internal word capitalized. Try to keep your class names simple and descriptive. Use whole words—avoid acronyms and abbreviations (unless the abbreviation is much more widely used than the long form such as URL or HTML. | class Raster;  class ImageSprite; |
| Interfaces | Interface name should be capitalized like class names. | interface RasterDelegate;  interface Storing; |
| Methods | Methods should be verbs in mixed case with the first letter lower case, with the first letter of each internal word capitalized. | run();  runFast();  getBackground(); |
| Variables | Except for variables, all instance, class and class constants are in mixed case with a lower case first letter. Internal words starts with capital letters. Variable names should not start with underscores \_, or dollar sign $ characters, even though both are allowed.  Variable name should be short yet meaningful. The choice of the variable name can be mnemonic that is designed to indicate to the casual observer the intent of the use. One character variable name should be avoided except for the temporary “throw away” variables. Common name for temporary variables are i j, k, m, n for integers, c, de, e for characters. |  |
| Constants | The name of variables declared class constants and ANSCII constants should be all upper-case with words separated by underscore (“\_”),  (ANSCII constants should be avoided for better debugging). | static final int MIN\_WIDTH = 4;  static final int MAX\_WIDTH = 9;  static final int GET\_THE\_CPU = 4; |

# Jfc Naming Conventions

JFC components consist of visual components and non-visual widgets. For components, the variables are post-fixed with the respective acronym for the components. In order to differentiate between the awt and swing components, the swing components are post-fixed with the letter “j”. For awt components, the letter “j” will not be there. The below mentioned is examples of coding conventions for JFC components.

|  |  |
| --- | --- |
| **Component Name** | **Acronym** |
| JTextField | jtxt<variable name> |
| JButton | jbtn<variable name> |
| Jtable | jtbl<variable name> |
| JTableHeader | jtbh<variable name> |
| JComboBox | jcmb<variable name> |
| JLabel | jlbl<variable name> |
| JList | jlst<variable name> |
| JRadioButton | jrbt<variable name> |
| JCheckBox | jcbx<variable name> |
| JTextArea | jtxa<variable name> |
| JFrame | jfrm<variable name> |
| JApplet | japl<variable name> |
| JInternalFrame | jifm<variable name> |
| JScrollPane | jspn<variable name> |
| JPopUpMenu | jpop<variable name> |
| JMenu | jmnu<variable name> |
| JMenuItem | jmni<variable name> |
| JMenuBar | jmbr <variable name> |
| JPanel | jpnl <variable name> |

# Member Function Visibility

For a good design where you minimize the coupling between classes, the general rule of thumb is to be as restrictive as possible when setting the visibility of a member function. If member function doesn’t have to be public then make it protected, and if it doesn’t have to be protected then make it private.

|  |  |  |
| --- | --- | --- |
| **Visibility** | **Description** | **Proper Usage** |
| Public | A public member function can be invoked by any other member function in any other object or class. | When the member function must be accessible by objects and classes outside of the class hierarchy in which the member function is defined. |
| Protected | A protected member function can be invoked by any member function in the class in which it is defined or any subclasses of that class. | When the member function provides behavior that is needed internally within the class hierarchy but not externally. |
| Private | A private member function can only be invoked by other member functions in the class in which it is defined, but not in the subclasses. | When the member function provides behavior that is specific to the class. Private member functions are often the result of refactoring, also known as reorganizing, the behavior of other member functions within the class to encapsulate one specific behavior. |
| <Default> | No visibility is indicated. This is called default or package visibility, and is sometimes referred to as friendly visibility. The member function is effectively public to all other classes within the same package, but private to classes external to the package. | This is an interesting feature, but be careful with its use. I use it when I’m building domain components, collections of classes that implement a cohesive business concept such as “Customer”, to restrict access to only the classes within the component/package. |

# Field Visibility

The fields not to be declared *public* for reasons of encapsulation, but it is better to go further to state that all fields should be declared *privat*e. When fields are declared *protected* there is the possibility of member functions in subclasses to directly access them, effectively increasing the coupling within a class hierarchy. This makes your classes more difficult to maintain and to enhance, therefore it should be avoided. Fields should never be accessed directly, instead accessor member functions (see below) should be used.

|  |  |  |
| --- | --- | --- |
| **Visibility** | **Description** | **Proper Usage** |
| Public | A public field can be accessed by any other member function in any other object or  class. | Do not make fields public. |
| Protected | A protected field can be accessed by any member function in the class in which it is declared or by any member functions  defined in subclasses of that class. | Do not make fields protected. |
| Private | A private field can only be accessed by  member functions in the class in which it is  declared, but not in the subclasses. | All fields should be private and be accessed  by getter and setter member functions (accessors). |

For fields that are not persistent (they will not be saved to permanent storage) you should mark them as either *static* or *transient.*

# Standards for Classes

# Class Visibility

Classes may have one of two visibilities: public or package (default). Public visibility is indicated with the keyword public and package visibility is not indicated (there is no keyword). Public classes are visible to all other classes whereas classes with package visibility are visible only to classes within the same package.

# Use package visibility for classes internal to a component.

With package visibility you hide classes within the package, effectively encapsulating them within your component.

# Use public visibility for the facades of components.

Components are encapsulated by façade classes, classes that implement the interface of the component and that route messages to classes internal to the component.

# Apply the “final” keyword sensibly

Use the keyword **final** to indicate that your class cannot be inherited from. This is a design decision on the part of the original developer, one that should not be taken lightly.

# Use Wild Cards When Importing Classes

The import statement allows the use of wildcards when indicating the names of classes. For example, the statement

**import java.awt.\*;**

brings in all of the classes in the package **java.awt** at once. Actually, that’s not completely true. What really happens is that every class that you use from the **java.awt** package will be brought into your code when it is compiled, classes that you do not use will not be.

# Alternative – Explicitly Specify Each Imported Class

Another approach is to fully qualify the name of the classes that your code uses, as shown in the example below:

**Examples:**

import java.awt.Color;

import java.awt.Button;

import java.awt.Container;

The problem with this approach is that it increases your maintenance burden – you need to keep your import list accurate whenever you add a new class (the compiler will force this on you) and whenever you stop using a class (you need to do this yourself).

# Servlets

All the previous sections hold true for servlets with the exception that servlets class names should end with the word ‘Servlet’.

**Examples:**

HelloWorldServlet

DbConnectServlet

# Code Conventions for JavaServer Pages

As JavaServer PagesTM (JSPTM) is becoming widely adopted in web-based applications, many JSP programmers and web developers embarking on developing and maintaining these applications face a dilemma like that of many JavaTM programmers, "How do we structure JSP code that is easier to read, write and maintain consistently?"

Hence a proposed set of standard conventions for writing JSPs (versions 1.1 and 1.2) that should be followed on a typical software project using web components is developed. It draws on the Code Conventions for the Java Programming Language as a template to identify various important elements that should be addressed in a coding conventions specification (relevant to JSP). In particular, it addresses file names and organization, indentation, comments, directives, declarations, scriptlets, expressions, white space, naming conventions, and programming practices.

The JavaServer Pages 2.0 Specification, while fully backwards compatible with version 1.2, allows for a script-free programming style (without declarations, scriptlets and expressions) and has a number of new features that are expected to evolve these conventions. Where possible, this article chooses conventions that will leverage the new JSP 2.0 features.

# File Name and Locations

File naming gives tool vendors and web containers a way to determine file types and interpret them accordingly. The following table lists our recommended file suffixes and locations.

|  |  |  |
| --- | --- | --- |
| **File Type** | **File Suffix** | **Recommended Location** |
| JSP | .jsp | <context root>/<subsystem path>/ |
| JSP fragment | .jsp | <context root>/<subsystem path>/ |
| .jspf | <context root>/WEB-INF/jspf/<subsystem path>/ |
| cascading style sheet | .css | <context root>/css/ |
| Javascript | .js | <context root>/js/ |
| HTML Page | .html | <context root>/<subsystem path>/ |
| web resource | .gif, .jpg etc | <context root>/images/ |
| tag library descriptor | .tld | <context root>/WEB-INF/tld/ |

There are a few notes when reading the table above. First, <context root> is the root of the context of the web application (the root directory inside a .war file). Second, <subsystem path> is used to provide refined logical grouping of dynamic and static web page contents. For a small web application, this may be an empty string.

Third, we use the term *JSP fragment* to refer to a JSP that can be included in another JSP. Note that in JSP 2.0, the term “JSP segment” is used instead as the term “JSP fragment” is overloaded. JSP fragments can use either .jsp or .jspf as a suffix, and should be placed either in /WEB-INF/jspf or with the rest of the static content, respectively. JSP fragments that are not complete pages should always use the .jspf suffix and should always be placed in /WEB-INF/jspf. Fourth, though the JSP specification recommends both .jspf and .jsp as possible extensions for JSP fragments, we recommend .jspf as .jsf might be used by the **JavaServer Faces™** specification.

Finally, it is in general a good practice to place tag library descriptor files and any other non-public content under WEB-INF/ or a subdirectory underneath it. In this way, the content will be inaccessible and invisible to the clients as the web container will not serve any files underneath WEB-INF/.

An optional welcome file's name, as declared in the welcome-file element of the deployment descriptor (web.xml), should be index.jsp if dynamic content will be produced, or index.html if the welcome page is static.

When internationalizing JSP files, we recommend that you group JSP pages into directories by their locale. For example, the US English version of index.jsp would appear under /en\_US/index.jsp whereas the Japanese version of the same file would appear under /ja\_JP/index.jsp.

# File Organization

A well-structured source code file is not only easier to read, but also makes it quicker to locate information within the file. This section introduces the structures for both JSP and tag library descriptor files.

### JSP FILE / JSP FRAGMENT FILE

A JSP file consists of the following sections in the order they are listed:

1. Opening comments
2. JSP page directive(s)
3. Optional tag library directive(s)
4. Optional JSP declaration
5. HTMLand JSP Code

# Opening Comments

A JSP file or fragment file begins with a server side style comment:

<%--

- Author(s):

- Date:

- Copyright Notice:

- @(#)

- Description:

--%>

This comment is visible only on the server side because it is removed during JSP translation. Within this comment are the author(s), the date, and the copyright notice of the revision, an identifier and a description about the JSP for web developers. The combination of characters "@(#) " is recognized by certain programs as indicating the start of an identifier. While such programs are seldom used, the use of this string does no harm. In addition, this combination is sometimes appended by "$Id$" for the identification information to be automatically inserted into the JSP by some version control programs. The Description part provides concise information about the purpose of the JSP. It does not span more than one paragraph.

In some situations, the opening comments need to be retained during translation and propagated to the client side (visible to a browser) for authenticity and legal purposes. This can be achieved by splitting the comment block into two parts, firstly the client side style comment:

<!--

- Author(s):

- Date:

- Copyright Notice:

--%>

and then a shorter server side style comment:

<%--

- @(#)

- Description:

--%>

# JSP Page Directive(s)

A JSP page directive defines attributes associated with the JSP at translation time. The JSP specification does not impose a constraint on how many JSP page directives can be defined in the same page. So the following two Code Samples are equivalent (except that the first example introduces two extra blank lines in the output):

**Code Sample 1:**

<%@ page session="false" %>

<%@ page import="java.util.\*" %>

<%@ page errorPage="/common/errorPage.jsp" %>

If the length of any directive, such as a page directive, exceeds the normal width of a JSP (80 characters), the directive is broken into multiple lines:

**Code Sample 2:**

<%@ page session="false"

import="java.util.\*"

errorPage="/common/errorPage.jsp"

%>

In general, Code Sample 2 is the preferred choice for defining the page directive over Code Sample 1. An exception occurs when multiple Java packages need to be imported into the JSP pages, leading to a very long import attribute:

<%@ page session="false"

import="java.util.\*,java.text.\*,

com.mycorp.myapp.taglib.\*,

com.mycorp.myapp.sql.\*, ..."

...

%>

In this scenario, breaking up this page directive like the following is preferred:

<%-- all attributes except import ones --%>

<%@ page

...

%>

<%-- import attributes start here --%>

<%@ page import="java.util.\*" %>

<%@ page import="java.text.\*" %>

...

Note that in general the import statements follow the local code conventions for Java. For instance, it may generally be accepted that when up to three classes from the same package are used, import should declare specific individual classes, rather than their package. Beyond three classes, it is up to a web developer to decide whether to list those classes individually or to use the ".\*" notation. In the former case, it makes life easier to identify an external class, especially when you try to locate a buggy class or understand how the JSP interacts with Java code. For instance, without the knowledge of the imported Java packages as shown below, a web developer will have to search through all these packages in order to locate a Customer class:

<%@ page import="com.mycorp.bank.savings.\*" %>

<%@ page import="com.thirdpartycorp.cashmanagement.\*" %>

<%@ page import="com.mycorp.bank.foreignexchange.\*" %>

...

In the latter case, the written JSP is neater but it is harder to locate classes. In general, if a JSP has too many import directives, it is likely to contain too much Java code. A better choice would be to use more JSP tags (discussed later in this article).

# Optional Tag Library Directive (s)

A tag library directive declares custom tag libraries used by the JSP. A short directive is declared in a single line. Multiple tag library directives are stacked together in the same location within the JSP body:

<%@ taglib uri="URI1" prefix="tagPrefix1" %>

<%@ taglib uri="URI2" prefix="tagPrefix2" %>

...

Just as with the page directive, if the length of a tag library directive exceeds the normal width of a JSP (80 characters), the directive is broken into multiple lines:

<%@ taglib

uri="URI2"

prefix="tagPrefix2"

%>

Only tag libraries that are being used in a page should be imported.

From JSP 1.2, it is highly recommended that the **JSP Standard Tag Library** be used in your web application to help reduce the need for JSP scriptlets in your pages. Pages that use JSTL are, in general, easier to read and maintain.

# Optional JSP Declaration (S)

JSP declarations declare methods and variables owned by the JSP. These methods and variables are no different from declarations in the Java programming language, and therefore the relevant code conventions should be followed. Declarations are preferred to be contained in a single <%! ... %> JSP declaration block, to centralize declarations within one area of the JSP body. Here is an example:

|  |  |
| --- | --- |
| **Disparate declaration blocks** | **Preferred declaration blocks** |
| <%! private int hitCount; %>  <%! private Date today; %>  ...  <%! public int getHitCount() {  return hitCount;  }  %> | <%!  private int hitCount;  private Date today;    public int getHitCount() {  return hitCount;  }  %> |

# HTML and JSP code

This section of the JSP holds the HTML body of the JSP and the JSP code, such JSP expressions, scriptlets, and JavaBeans instructions.

#### Tag Library Descriptor

A tag library descriptor (TLD) must begin with a proper XML declaration and the correct DTD statement. For example, a JSP 1.2 TLD must begin with:

<?xml version="1.0" encoding="ISO-8859-1" ?>

<!DOCTYPE taglib

PUBLIC "-//Sun Microsystems, Inc.//DTD JSP Tag Library 1.2//EN"

"http://java.sun.com/dtd/web-jsptaglibrary\_1\_2.dtd">

This is immediately followed by a server side style comment that lists the author(s), the date, the copyright, the identification information, and a short description about the library:

<!--

- Author(s):

- Date:

- Copyright Notice:

- @(#)

- Description:

-->

The rules and guidelines regarding the use of these elements are the same for those defined for JSP files/fragment files.

The rest of the tag library file consists of in the following in the order they appear below:

Optional declaration of one tag library validator

Optional declaration of event listeners

Declaration of one or more available tags

It is recommended that you always add the following optional sub-elements for the elements in a TLD. These sub-elements provide placeholders for tag designers to document the behavior and additional information of a TLD, and disclose them to web component developers

|  |  |  |
| --- | --- | --- |
| **TLD Element** | **JSP 1.2 Recommended Sub-element** | **JSP 1.1 Recommended Sub-element** |
| attribute (JSP 1.2) | Description |  |
| init-param (JSP 1.2) | Description |  |
| tag | display-name, description, example | name, info |
| taglib | uri, display-name, description | uri, info |
| validator (JSP 1.2) | Description |  |
| variable (JSP 1.2) | Description |  |

#### Indentation

Indentations should be filled with space characters. Tab characters cause different interpretation in the spacing of characters in different editors and should not be used for indentation inside a JSP. Unless restricted by particular integrated development environment (IDE) tools, a unit of indentation corresponds to 4 space characters. Here is an example:

<myTagLib:forEach var="client" items="${clients}">

<myTagLib:mail value="${client}" />

</myTagLib:forEach>

A continuation indentation aligns subsequent lines of a block with an appropriate point in the previous line. The continuation indentation is in multiple units of the normal indentation (multiple lots of 4 space characters):

<%@ page attribute1="value1"

attribute2="value2"

...

attributeN="valueN"

%>

#### Indentation of Scripting Elements

When a JSP scripting element (such as declaration, scriptlet, expression) does not fit on a single line, the adopted indentation conventions of the scripting language apply to the body of the element. The body begins from the same line for the opening symbol of the element <%=, and from a new line for the opening symbol <%=. The body is then terminated by an enclosing symbol of the element (%>) on a separate line. For example:

<%= (Calendar.getInstance().get(Calendar.DAY\_OF\_WEEK)   
        = Calendar.SUNDAY) ?   
    "Sleep in" :   
    "Go to work"   
%>

The lines within the body not containing the opening and the enclosing symbols are preceded with one unit of normal indentation (shown as   in the previous example) to make the body distinctively identifiable from the rest of the JSP.

#### Compound Indentation with JSP, HTML and Java

Compound indentation, for JSP elements intermingled with Java scripting code and template text (HTML), is necessary to reduce the effort of comprehending a JSP source file. This is because the conventional normal indentation might make seeing the JSP source file difficult. As a general rule, apply an extra unit of normal indentation to every element introduced within another one. Note that this alters the indentations of the final output produced for the client side to render for display. The additional indentations, however, are usually ignored (by the browser) and have no effect on the rendered output on the browser. For instance, adding more space characters before a <TABLE> tag does not change the position of a table. So, applying this convention for indentation makes this looks nicer:

<table>

<% if { tableHeaderRequired ) { %>

<tr>

<th>Last Name</th>

<th>First Name</th>

</tr>

<% } %>

<c:forEach var="customer" items="${customers}">

<tr>

<td><c:out value="${customer.lastName}"/></td>

<td><c:out value="${customer.firstName}"/></td>

</tr>

</c:forEach>

</table>

than this:

<table>

<% if { tableHeaderRequired ) { %>

<tr>

<th>Last Name</th>

<th>First Name</th>

</tr>

<% } %>

<c:forEach var="customer" items="${customers}">

<tr>

<td><c:out value="${customer.lastName}"/></td>

<td><c:out value="${customer.firstName}"/></td>

</tr>

</c:forEach>

</table>

# Comments

Comments are used sparingly to describe additional information or purposes of the surrounding code. Here we look at two types for JSP files: JSP and client side comments

#### JSP Clients

JSP comments (also called server side comments) are visible only on the server side (that is, not propagated to the client side). Pure JSP comments are preferred over JSP comments with scripting language comments, as the former is less dependent on the underlying scripting language, and will be easier to evolve into JSP 2.0. The following table illustrates this

|  |  |  |
| --- | --- | --- |
| **Line** | **JSP scriptlet with scripting language comment** | **Pure JSP comment** |
| Single | <% /\*\* ... \*/ %>  <% /\* ... \*/ %>  <% // ... %> | <%-- ... --%> |
| Multiple | <%  /\*  \*  ...  \*  \*/  %> | <%--  -  ...  -  -- %> |
| <%  //  //  ...  //  %> |

#### Client Side Comments

Client side comments (<!-- ... -->) can be used to annotate the responses sent to the client with additional information about the responses. They should not contain information about the behavior and internal structure of the server application or the code to generate the responses.

The use of client side comments is generally discouraged, as a client / user does not need or read these kinds of comments directly in order to interpret the received responses. An exception is for authenticity and legality purposes such as the identification and copyright information as described above. Another exception is for HTML authors to use a small amount of HTML comments to embody the guidelines of the HTML document structures, for example

<!-- toolbar section -->

...

<!-- left-hand side navigation bar -->

...

<!-- main body -->

...

<!-- footer -->

...

#### Multiline Comment Block

A multiline comment block, be it JSP or client side, is decorated with the dash character "-". In the XML specification, the double-dash string "--" is not allowed within an XML comment block. Thus, for compatibility and consistency with this specification, no double-dash string is used to decorate comment lines within a multiline comment block. The following table illustrates this preference using a client side comment block:

|  |  |
| --- | --- |
| **Preferred** | **Non – XML Compliant** |
| <!--  - line 1  - line 2  ...  --> | <!--  -- line 1  -- line 2  ...  --> |

# JSP declarations

As per the Java code convention, declarations of variables of the same types should be on separate lines:

|  |  |
| --- | --- |
| **Not recommended** | **Recommended** |
| <%! private int x, y; %> | <%! private int x; %> <%! private int y; %> |

JavaBeans should not be declared and instantiated using JSP declarations but instead should use the <jsp:useBean> action tag.

In general, JSP declarations for variables are discouraged as they lead to the use of the scripting language to weave business logic and Java code into a JSP which is designed for presentation purposes, and because of the overhead of managing the scope of the variables.

# JSP SCriptlets

Where possible, avoid JSP scriptlets whenever tag libraries provide equivalent functionality. This makes pages easier to read and maintain, helps to separate business logic from presentation logic, and will make your pages easier to evolve into JSP 2.0-style pages (JSP 2.0 supports but deemphasizes the use of scriptlets). In the following examples, for each data type representation of the customers, a different scriptlet must be written:

customers as an array of Customers

<table>  
 <% for ( int i=0; i<customers.length; i++ ) { %>  
 <tr>  
 <td><%= customers[i].getLastName() %></td>  
 <td><%= customers[i].getFirstName() %></td>  
 </tr>  
 <% } %>  
 </table>

customers as an Enumeration

<table>  
 <% for ( Enumeration e = customers.elements();  
 e.hasMoreElements(); ) {   
 Customer customer = (Customer)e.nextElement();  
 %>  
 <tr>  
 <td><%= customer.getLastName() %></td>  
 <td><%= customer.getFirstName() %></td>  
 </tr>  
 <% } %>  
 </table>

However, if a common tag library is used, there is a higher flexibility in using different types of customers. For instance, in the JSP Standard Tag Library, the following segment of JSP code will support both array and Enumeration representations of customers:

<table>  
 <c:forEach var="customer" items="${customers}">  
 <tr>  
 <td><c:out value="${customer.lastName}"/></td>  
 <td><c:out value="${customer.firstName}"/></td>  
 </tr>  
 </c:forEach>  
 </table>

In the spirit of adopting the model-view-controller (MVC) design pattern to reduce coupling between the presentation tier from the business logic, JSP scriptlets should not be used for writing business logic. Rather, JSP scriptlets are used if necessary to transform data (also called "value objects") returned from processing the client's requests into a proper client ready format. Even then, this would be better done with a front controller servlet or a custom tag. For example, the following code fetches the names of customers from the database directly and displays them to a client:

<%  
 // NOT RECOMMENDED TO BE DONE AS A SCRIPTLET!  
  
 Connection conn = null;  
 try {  
 // Get connection  
 InitialContext ctx = new InitialContext();  
 DataSource ds = (DataSource)ctx.lookup("customerDS");  
 conn = ds.getConnection();  
  
 // Get customer names  
 Statement stmt = conn.createStatement();  
 ResultSet rs = stmt.executeQuery("SELECT name FROM customer");  
  
 // Display names  
 while ( rs.next() ) {  
 out.println( rs.getString("name") + "<br>");  
 }  
 } catch (SQLException e) {  
 out.println("Could not retrieve customer names:" + e);  
 } finally {  
 if ( conn != null )   
 conn.close();  
 }  
%>

The following segment of JSP code is better as it delegates the interaction with the database to the custom tag myTags:dataSource which encapsulates and hides the dependency of the database code in its implementation:

<myTags:dataSource  
 name="customerDS"  
 table="customer"   
 columns="name"  
 var="result" />  
<c:forEach var="row" items="${result.rows}">  
 <c:out value="${row.name}" />  
 <br />  
</c:forEach>

result is a scripting variable introduced by the custom tag myTags:dataSource to hold the result of retrieving the names of the customers from the customer database. The JSP code can also be enhanced to generate different kinds of outputs (HTML, XML, WML) based on client needs dynamically, without impacting the backend code (for the dataSource tag). A better option is to delegate this to a front controller servlet, which performs the data retrieval, and provide the results to the JSP through a request-scoped attribute. See the Enterprise section of Java BluePrints for an example.

In summary:

1. JSP scriptlets should ideally be non-existent in the JSP so that the JSP is independent of the scripting language, and business logic implementation within the JSP is avoided.
2. If not possible, use value objects (JavaBeans) for carrying information to and from the server side, and use JSP scriptlets for transforming value objects to client outputs
3. Use custom tags (tag handlers) whenever available for processing information on the server side.

# JSP expressions

JSP Expressions should be used just as sparingly as JSP Scriptlets. To illustrate this, let's look as the following three examples, which accomplish equivalent tasks:

Example 1 (with explicit Java code):

    <%= myBean.getName() %>

Example 2 (with JSP tag):

    <jsp:getProperty name="myBean" property="name" />

Example 3 (with JSTL tag):

    <c:out value="${myBean.name}" />

Example 1 assumes that a scripting variable called myBean is declared. The other two examples assume that myBean is a scoped attribute that can be found using PageContext.findAttribute(). The second example also assumes that myBean was introduced to the page using <jsp:useBean

Of the three examples, the JSTL tag example is preferred. It is almost as short as the JSP expression, it is just as easy to read and easier to maintain, and it does not rely on Java scriptlets (which would require the web developer to be familiar with the language and the API calls). Furthermore, it makes the page easier to evolve into JSP 2.0-style programming, where the equivalent can be accomplished by simply typing ${myBean.name} in template text. Whichever choice is adopted, it should be agreed on amongst web developers and consistent across all produced JSPs in the same project.  It should be noted that the JSTL example is actually slightly different in that it gets the value of myBean from the page context instead of from a local Java scripting variable.

Finally, JSP expressions have preference over equivalent JSP scriptlets which rely on the syntax of the underlying scripting language. For instance,

<%= x %>

is preferred over

<% out.print( x ); %>

# White Space

White space further enhances indentation by beautifying the JSP code to reduce comprehension and maintenance effort. In particular, blank lines and spaces should be inserted at various locations of a JSP file where necessary.

# Blank Lines

Blank lines are used sparingly to improve the legibility of the JSP, provided that they do not produce unwanted effects on the outputs. For the example below, a blank line inserted between two JSP expressions inside an HTML <PRE> block call causes an extra line inserted in the HTML output visible in the client's browser. However, if the blank line is not inside a <PRE> block, the effect is not visible in the browser's output.

|  |  |
| --- | --- |
| **JSP statements** | **HTML output to client** |
| <pre> <%= customer.getFirstName() %> <%= customer.getLastName() %> </pre> | Joe Block |
| <pre> <%= customer.getFirstName() %>   <%= customer.getLastName() %> </pre> | Joe  Block |
| <%= customer.getFirstName() %>  <%= customer.getLastName() %> | Joe Block |

# Blank Spaces

A white space character (shown as  ) should be inserted between a JSP tag and its body. For instance, the following

<%= customer.getName() %>

is preferred over

<%=customer.getName()%>

There should also be space characters separating JSP comment tags and comments:

<%--   
 - a multi-line comment broken into pieces, each of which  
 - occupying a single line.  
 --%>  
<%-- a short comment --%>

# Naming Conventions

Applying naming conventions makes your web component elements easier to identify, classify and coordinate in projects. In this section, we will look at these conventions specific to JSP.

# JSP Names

A JSP (file) name should always begin with a lower case letter. The name may consist of multiple words, in which case the words are placed immediately adjacent and each word commences with an upper-case letter. A JSP name can be just a simple noun or a short sentence. A verb-only JSP name should be avoided, as it does not convey sufficient information to developers. For example:

perform.jsp

is not as clear as

performLogin.jsp

In the case of a verb being part of a JSP name, the present tense form should be used as an action by way of backend processing is implied:

showAccountDetails.jsp

is preferred over

showingAccountDetails.jsp

# TAG names

The naming conventions for tag handlers and associated classes are shown below:

|  |  |
| --- | --- |
| **Description** | **Class Name** |
| XXX tag extra info (extending from javax.servlet.jsp.tagext.TagExtraInfo) | XXXTEI |
| XXX tag library validator (extending from javax.servlet.jsp.tagext.TagLibraryValidator) | XXXTLV |
| XXX tag handler interface (extending from javax.servlet.jsp.tagext.Tag/IterationTag/BodyTag) | XXXTag |
| XXX tag handler implementation | XXXTag |

In addition, tag names must not violate the naming conventions of class and interface as specified in the relevant code convention for Java.

To further distinguish a tag relevant class from other classes, a package suffix, tags, or taglib, can be applied to the package name of the class, for example:

com.mycorp.myapp.tags.XXXTag

# TAG prefix names

A tag prefix should be a short yet meaningful noun in title case, and the first character in lower case. A tag prefix should not contain non-alphabet characters. Here are some examples:

|  |  |
| --- | --- |
| **Example** | **OK ?** |
| Mytaglib | no |
| myTagLib | yes |
| MyTagLib | no |
| MyTagLib1 | no |
| My\_Tag\_Lib | no |
| My$Tag$Lib | no |

# JSP pages in XML SYNTAX

JSP provides two distinct syntaxes: a 'standard syntax' for writing JSP pages and an 'XML syntax' for writing JSP as an XML document. JSPs that are written using the standard syntax are referred to as 'JSP Pages'. JSPs that are written using the XML syntax are referred to as 'JSP Documents'. This article primarily addresses JSP Pages, but many of the concepts can be applied to JSP Documents as well. Use of JSP Documents is expected to increase as XML becomes more prevalent, and to address this the JSP 2.0 specification will introduce a much friendlier XML syntax.

It should be noted that the XML *syntax* used to author JSPs is distinct from and is often confused with the XML *view* of a JSP. The page author uses either the standard or the XML *syntax* to author a JSP. The container then translates the JSP into its XML *view*, which is exposed to Tag Library Validators.

# JSP document structure

JSP documents have the following basic structure:

<? xml version="1.0" ?>  
 <!--   
 - Author(s):   
 - Date:  
 - Copyright Notice:  
 - @(#)  
 - Description:  
 -->  
 <jsp:root xmlns:jsp="http://java.sun.com/JSP/Page"  
 xmlns:prefix1="URI-for-taglib1"  
 xmlns:prefix2="URI-for-taglib2"  
 version="1.2">  
 JSP Document ...  
 </jsp:root>

The first line is an optional XML Prolog that defines the page as an XML document. After the optional prolog comes the comments for the document. The element <jsp:root> defines this as a JSP Document and must appear as the root element. The jsp namespace must be imported, and all tag libraries must be imported using this root element. The version attribute is required and specifies which version of JSP is being used. The actual content of the JSP document appears as sub-elements of the <jsp:root> element. Standard XML indentation rules should be applied consistently across the document, using 4 spaces as a single indentation unit.

A JSP Document must be a well-formed XML document, so some elements, such as <% %> must be replaced by their XML equivalent, such as <jsp:scriptlet />. See the JSP Specification for details.

# XML Comments

The JSP Specification is unclear about whether XML-style comments are stripped on output, so to be safe if a comment is intended to reach the client, it should be enclosed in a <jsp:text> node, as follows:

...  
 <jsp:text><![CDATA[  
 <!--  
 - Multiline comment  
 - to be sent to client.  
 -->  
 ]]></jsp:text>  
 ...

# Java code in JSP documents

When writing Java code inside declarations, scriptlets, and expressions, a CDATA element should be used only when necessary to ensure your code does not break the document structure.

...  
 <jsp:scriptlet>  
 for( int level = 0; level < 3; level++ ) {  
 </jsp:scriptlet>  
 <tr>  
 <td>  
 <jsp:expression><![CDATA[  
 "<h" + level + ">Text</h" + level + ">"  
 ]]></jsp:expression>  
 </td>  
 </tr>  
 <jsp:scriptlet>  
 }  
 </jsp:scriptlet>  
 ...

Unlike those in the standard syntax, XML indentation rules should be followed regardless of the contents of an element.

# Programming Practice

In general, avoid writing Java code (declarations, scriptlets and expressions) in your JSP pages, for the following reasons:

1. Syntax errors in Java code in a JSP page are not detected until the page is deployed.
2. Syntax errors in tag libraries and servlets, on the other hand, are detected prior to deployment.
3. Java code in JSP pages is harder to debug.
4. Java code in JSP pages is harder to maintain, especially for page authors who may not be Java experts.
5. It is generally accepted practice not to mix complex business logic with presentation logic. JSPs are primarily intended for presentation logic.
6. Code containing Java code, HTML and other scripting instructions can be hard to read.
7. JSP 2.0 is de-emphasizing scriptlets in favor of a much simpler expression language. It will be easier to evolve your JSPs to JSP 2.0-style programming if Java code is not used in your pages.

# JAVA Beans initialization

JSP provides a convenient element to initialize all PropertyDescriptor-identified properties of a JavaBeanTM, for instance:

<jsp:setProperty name="bankClient" property="\*"/>

However, this should be used with caution. First, if the bean has a property, say, amount, and there is no such parameter (amount) in the current ServletRequest object or the parameter value is "", nothing is done: the JSP does not even use null to set that particular property of the bean. So, whatever value is already assigned to amount in the bankClient bean, is unaffected. Second, non-elementary properties that do not have PropertyEditors defined may not be implicitly initialized from a String value of the ServletRequest object and explicit conversion may be needed. Third, malicious users can add additional request parameters and set unintended properties of the bean, if the application is not carefully designed.

If you still prefer to use property="\*" in the jsp:setProperty tag for the purpose of producing neater code, then we recommend that you add a comment preceding the jsp:setProperty tag about parameters expected to be present in the ServletRequest object to initialize the bean. So, in the following example, from the comment we know that both firstName and lastName are required to initialize the bankClient bean:

<%--  
 - requires firstName and lastName from the ServletRequest   
 --%>  
<jsp:setProperty name="bankClient" property="\*" />

# JSP Implicit TAGs

Direct use of JSP implicit objects to gain references to these objects rather than API calls is preferred. So, instead of using

getServletConfig().getServletContext().getInitParameter("param")

to access the initialization parameter as provided by the ServletContext instance, one can make use of the readily available implicit object:

application.getInitParameter("param")

In the case that only the value of an initialization parameter is outputted, it would be even better to use JSTL to access the initialization parameter:

<c:out value="${initParam['param']}" />

## quoting

The uniform use of quoting is adopted. Two double-quote characters “instead of two apostrophe characters’ should bind quotations.

|  |  |
| --- | --- |
| **Non-uniform quoting** | **Preferred quoting** |
| <%@ page import=**'**javabeans.\***'**%> <%@ page import="java.util.\*" %> | <%@ page import="javabeans.\*" %> <%@ page import="java.util.\*" %> |

An exception is when apostrophes are needed, for example when double-quote characters are required within the scripting language:

<jsp:include page='<%= getFoodMenuBar("Monday") %>' />

# Using Custom Tags

If a custom tag does not have a body content, the content should be declared explicitly with empty (rather than defaulting to "JSP") like this in the tag library descriptor:

<tag>  
 <name>hello</name>  
 <tag-class>com.mycorp.util.taglib.HelloTagSupport</tag-class>  
 <body-content>empty</body-content>  
 ...  
</tag>

This tells the JSP container that the body content must be empty rather than containing any JSP syntax to be parsed. The effect is to eliminate unnecessarily allocation of resources for parsing of empty body contents.

Empty tags should be in short XML elements, rather than using opening-closing XML element pairs to improve readability. So,

<myTag:hello />

is preferred over

<myTag:hello></myTag:hello>

## use of tagextrainfo and taglibraryvalidator

Sometimes, the valid ways to use a tag library cannot be expressed using the TLD alone. Then, a TagExtraInfo class or a TagLibraryValidator class should be written and registered in the TLD so that errors in tag library can be caught at translation time.

## use of javascript

JavaScriptTM should be independent of particular features of browser types in order for the scripts to run properly.

Where it makes sense, it is a good idea to keep JavaScript code in individual files separate from JSP bodies, and use a statement like the following to import the JavaScript into the JSP bodies:

<script language=javascript src="/js/main.js">

This improves the chance for the JavaScript code to be reused, maintains the consistent behavior of the JavaScript code across multiple JSPs, and reduces the complexity of JSPs.

## cascading style sheet

Use cascading style sheets to centralize control of common characteristics of headings, tables, and so on. This improves the consistency of presentation to the users and reduces maintenance effort and the code size of the JSP pages. So, instead of embedding the style information in the HTML tags like the one below:

<H1><FONT color="blue">Chapter 1</FONT></H1>  
...  
<H1><FONT color="blue">Chapter 2</FONT></H1>  
...

Define the style information in a single style sheet myJspStyle.css which contains:

H1 { color: blue }

And apply the style sheet to the JSP page:

<link rel="stylesheet" href="css/myJspStyle.css" type="text/css">  
...  
<H1>Chapter 1</H1>  
...  
<H1>Chapter 2</H1>

## use of composite view pattern

When a JSP requires a certain and complex structure which may also repeat in other JSPs, one way to handle this is to break it up into pieces, using the Composite View pattern (the Patterns section of Java Blueprints). For instance, a JSP sometimes has the following logical layout in its presentation structure:

|  |  |
| --- | --- |
| Header | |
| menu bar | main body |
| footnote |
| Footer | |

In this manner, this composite JSP can be divided into different modules, each realized as a separate JSP. Constituent JSPs can then be placed in appropriate locations in the composite JSP, using translation-time or request-time include JSP tags. In general, when static include directives are used to include a page that would not be requested by itself, remember to use the .jspf extension and place the file in the /WEB-INF/jspf/ directory of the Web application archive (war). For example:

<%@ include file="/WEB-INF/jspf/header.jspf" %>  
...  
<%@ include file="/WEB-INF/jspf/menuBar.jspf" %>  
...  
<jsp:include page="<%= currentBody %>" />  
...  
<%@ include file="/WEB-INF/jspf/footnote.jspf" %>  
...  
<%@ include file="/WEB-INF/jspf/footer.jspf" %>

**Revision History**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Version no.** | **Date** | **Author** | **Description** | **Reviewed by** | **Approved by** |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |