

Java Programming Best Practices and Conventions

This document should be used as a guideline for programming standards in Java. The document is not meant to provide a full detailed description of programming practices, but rather it concentrates on my programming inconsistencies in coding, and the not so obvious practices.

Files and project structure

\projects (parent folder for all projects)

\[project name] (parent folder for whole project name)

README

build.xml

build.properties

ftpupload.scp

ftptest.scp

\build (auto generated build directory)

\dist (auto generated distribution directory)

\ejb (Enterprise Java Beans storage)

\com

\[project name]

\package name (ejb location)

\META-INF (jboss.xml, ejb-jar.xml, jaws.xml)

\web (all presentation web files (.jsp, .css, .html etc.)

\images (images)

\servlets

\com

\[project name]

\servlets (servlet package location)

\WEB-INF (web.xml, jboss-web.xml)

Documentation

General Points

- Use comments, i.e. READ comments and write them.
- Document why something is being done – most of the time the problem is not understanding the code but why it is written.
- Keep comments simple
- Write documentation before you code

Documentation Style

There are three types of comments:-

- Documentation comments (javadoc) `/** */`
- C-Style Comments `/* */`
- Inline comments `//`

Usage:-

Documentation Comments	Class and Method documentation
C-Style Comments	Old Code Include the date when code was commented out, programmer doing the edit and when can the old code be removed, eg:- <code>/* This code was commented out on 15/11/2004 by Michael to rectify calculation. Delete after two months if everything is ok ... [source code] */</code>
Inline Comments	Implementation comments Special Comments Use XXX in a comment to flag something that is bogus but works. Use FIXME to flag something that is bogus and broken. Use TODO for pending things

What to document

- **Old Code** – Before deleting code especially critical sections in production systems document the old code and delete after testing is complete.
- **The processing order** - If there are statements in your code that must be executed in a defined order then you should ensure that this fact gets documented.
- **Why, as well as what, the code does** – You can always look at a piece of code and figure out what it does, but for code that isn't obvious you can rarely determine why it is done that way. For example, you can look at a line of code and easily determine that a 5% discount is being applied to the total of an order. That is easy. What isn't easy is figuring out WHY that discount is being applied. Obviously there is some sort of business rule that says to apply the discount, so that business rule should at least be referred to in your code so that other developers can understand why your code does what it does.
- **Difficult or complex code**
- **Methods and Classes**

Documentation Presentation

Class documentation

- **The purpose of the class** – Developers need to know the general purpose of a class so they can determine whether or not it meets their needs. I also make it a habit to document any good things to know about a class, for example is it part of a pattern or are there any interesting limitations to using it.
- **Known bugs** – If there are any outstanding problems with a class they should be documented so that other developers understand the weaknesses/difficulties with the class. Furthermore, the reason for not fixing the bug should also be documented. Note that if a bug is specific to a single member function then it should be directly associated with the member function instead.
- **The development/maintenance history of the class** – It is common practice to include a history table listing dates, authors,

and summaries of changes made to a class. The purpose of this is to provide maintenance programmers insight into the modifications made to a class in the past, as well as to document who has done what to a class.

- **Date and Author**

Method documentation

- **Like Class documentation**
- **Documenting Parameters** – Parameters to a member function are documented in the header documentation for the member function using the *javadoc* `@param` tag. You should describe:
 - **What it should be used for**
 - **Any restrictions or preconditions.** If the full range of values for a parameter is not acceptable to a member function, then the invoker of that member function needs to know.

Naming

Packages – Names of packages should have the format
com.[project name].Package name e.g. com.eiffel.form

Class Naming – Full names with each word capitalised – RasterImage,

Interfaces – Like class names

Methods, Variable names – Small initial and subsequent caps

Constants – All caps with underscore

Programming practices

Try to initialise variables where they are declared.

Public Properties - Don't make any instance or class variable public without good reason. Often, instance variables don't need to be explicitly set or gotten—often that happens as a side effect of method calls. One example of appropriate public instance variables is the case where the class is essentially a data structure, with no behavior.

For loop counters use variables **i, j, k**

Numerical constants (literals) should not be coded directly, except for -1, 0, and 1, which can appear in a `for` loop as counter values.

Parentheses - It is generally a good idea to use parentheses liberally in expressions involving mixed operators to avoid operator precedence problems.

```
if (a == b && c == d) // AVOID!  
if ((a == b) && (c == d)) // USE
```

Checklist

General

- Keep comment simple
- Document why and how
- Comment old code in c-style
- Implementation comments with inline
- Documentation comments with javadoc

Class documentation

- Purpose of class
- Known bugs
- Development History
- Date
- Author

Method documentation

- Parameter documentation
- Usage
- Any restrictions or preconditions

Naming

- Packages – com.[project name].package name
- Class naming – like RasterImage
- Interface naming – like RasterImage
- Methods, variables – like rasterImage
- Constants – like RASTER_IMAGE_SIZE

Databases Best Practices and Conventions

Project Structure

- `\my documents\sql\[project name]` – Keeps all SQL files pertaining to project
- `\project\data` – Keeps data files pertaining to project

Naming Convention

Table – Underscore, singular
e.g. `form_user`, `form_passenger_detail`

Column – Underscore, singular
e.g. `passenger_name`, `ticketing_date`

Practices

- Always use table column names when inserting and selecting this will keep the code working if the table structure changes.
- Always add Foreign keys and constraints as much as possible to validate input from user.
- Split names from surnames
- When applicable write all data in caps. This makes searching easier.
- Always prefix table name by schema name (`authority.form_user` instead of `form_user`)
- Avoid using like wild characters at the beginning of a word (like `'%pple'`)
- Always retrieve connection from connection pool
- Free resources and discard connection variables in finally part of try
- Use PreparedStatements to insert data, this will prevent quotes problem

J2EE Best Practices and Conventions

Practices

- Entity bean should represent and store information not implement business logic
- Session bean should contain all business logic
- Access cycle web -> servlet -> session bean -> entity bean

- When accessing objects from the request instance always check for null

References

Main

Java Code Conventions

<http://java.sun.com/docs/codeconv/html/CodeConvTOC.doc.html>

Coding Standards for Java from AmbySoft

<http://www.ambysoft.com/javaCodingStandards.html>

Others

<http://www.geosoft.no/development/javastyle.html>

<http://gee.cs.oswego.edu/dl/html/javaCodingStd.html> - Doug Lea

<http://mindprod.com/unmain.html>

<http://www.chimu.com/publications/javaStandards/index.html>

<http://java.sun.com/j2se/javadoc/writingdoccomments/>

Code Complete, Steve McConnell - Microsoft Press