Keith R. Bennett's Technical Blog

Sun Java Coding Conventions Revisited

By keithrbennett on March 6th, 2010

The Sun Java Code Conventions document, written in 1997, and available at http://java.sun.com/docs/codeconv/, continues to be a valuable resource for Java programmers. In addition to the nuts and bolts of formatting and the like, it includes some great wisdom, such as:

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.

I've recently been asked to participate on a committee that will come up with a set of coding standards. These standards will be used by several teams, so it's especially important that they be good and not overly restrictive. The Sun conventions are a reasonable place we may start. On the whole, I think they're great, but I do have some reservations about a few points. Below are some notes regarding the Sun conventions, listed by section in the original document to which they refer. Quotes from the Sun document are indicated in italics, except for source code. I invite your feedback. This article may be modified based on your comments or my own "clearer thinking and better information".

3.1.1 Beginning Comments

All source files should begin with a c-style comment that lists the programmer(s), the date, a
copyright notice, and also a brief description of the purpose of the program. For example:

```
/*
* Classname
```

```
*
* Version info
*
* Copyright notice
*/
```

Although the paragraph mentions a date, the code refers to a version. In any case, regarding the inclusion of either one in the source code, I believe this should be flexible, taking into account team practices. On most projects I've worked on in recent years, this inclusion has not been necessary, and requiring it would have been a burden and would have degraded the signal to noise ratio of the code.

In addition, including version information usually embeds dependency on a specific version control system in the source code, and thereby increases the cost of switching version control systems. Even worse is manual modification of version numbers, which is highly error prone (I hope no one actually does this). In my experience in recent years, source code is rarely printed, and developers rarely have confusion over whether they are working on the head of the trunk or some other version. In the unusual case that the developer has any confusion about the version number, he/she can consult the IDE or version control system.

Regarding the author information (which is included in the paragraph but not the example code), I have mixed feelings about that. It's nice to know who to consult, but on the other hand, it's a very rough and potentially outdated piece of information. Much more information can be found, quite easily, by doing a version control annotate. Also, these author comments are biased toward the original author. Does anyone have any hard and fast rules about at what point a new author should be added? It may be done inconsistently within a team, or over time. So the author information included in the header may not be as accurate as it seems.

4. Indentation

I suggest spaces, never tab characters; the tab setting of 8 is in that case irrelevant. On the other hand, if there is any possibility of a tab character sneaking in, I'd suggest setting tab width to 4.

4.1. Line Length

The line length limit of 80 is an anachronism, a relic of text mode terminals and days gone by when printing source code was common. These days, though, it is no longer practical. Source code is rarely printed,

lines are longer, and a short line maximum means more line continuations, which means code that is less readable. There should be a maximum, but it should be higher than 80. It should take into account typical use cases of developers (e.g. screen size/resolution and window width), but also accommodate less common cases (e.g. developers who use larger font sizes due to vision issues).

6.2 Placement [of Declarations], 6.3 Initialization

Put declarations only at the beginning of blocks.

I disagree with this. I think putting a declaration as close as possible to (before) its use:

- communicates more information to the reader, that is, that the variable is not used above the declaration;
- is an indicator to the developer that there may be an opportunity or a need to extract the inner code into its own method; and
- makes that extraction simpler to accomplish.

7.5 For Statements, 7.6 While 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);
```

I disagree; I believe the semicolon should be on the following line, or empty braces used:

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

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

This communicates the bounds of the for statement far more clearly; the other way, lines following the one line *for* statement can easily be confused as belonging to the loop.

The same applies to while loops.

7.8 Switch Statements

A switch statement should have the following form:

```
switch (condition) {
case ABC:
....statements;
..../* falls through */
case DEF:
....statements;
....break;
default:
....statements;
....break;
}
```

I think indenting the case relative to the switch should also be permitted:

```
switch(something) {
....case FOO:
........do_something();
```

This makes it easier to visually locate the beginning of the switch statement.

8.1 Blank Lines

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

- *Between methods*

The Sun coding conventions were written at a time when it was still not uncommon to print source files. Now, however, source files are rarely printed. The interest of legibility no longer needs to be tempered by the need for economy of vertical whitespace.

I suggest permitting (or even suggesting) two blank lines as method separators. This more clearly communicates that the degree of distinction between the methods exceeds the degree of distinction between sections of a method.

9. Naming Conventions

Variable names should be short yet meaningful.

I'd like to add:

"In cases where making a variable name longer eliminates the need for an explanatory comment, this guideline may be relaxed somewhat."

10.5.4 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.

I think "KLUDGE" is more expressive and universal than "XXX". "KLUDGE" can successfully be looked up on Google. "XXX" can be also, but you wouldn't find anything about software. 😉

11.1 Java Source Code Example

The following example shows how to format a Java source file containing a single public class.

```
/*
* %W% %E% Firstname Lastname
//...
```

The tokens are probably meant for substitution by a version control system. I suggest *not* putting anything in source code that is version control system specific, unless the benefits are huge. Most of us are using old version control systems, and it would be nice to keep the cost of upgrading to a better one low. Sun may not have been suggesting the use of these tokens, but I think it might be helpful to explicitly caution the reader about the downside of doing so.

That's what I think, but my beliefs are a function of my finite knowledge and experience. There may be other ways of viewing these issues. What do *you* think?

Categorized under: Uncategorized.

Tagged with: code conventions, code standards, coding conventions, coding standards, java, sun.

2 Responses to "Sun Java Coding Conventions Revisited"

Charles Calvert says: June 12, 2010 at 7:36 pm

I take the opposite approach to indentation: use tabs, not spaces. Each tab character is a level of indentation. My reason for using tabs is this: Many programmers have different preferences when it comes to the number of character positions to indent a block. If you use tabs to indent, you can configure your editor to represent that tab as N spaces. I can have my editor show 4 character positions, while Jack can have his show 2. Note that I said “represent”, not “convert”. The tab remains unchanged in the file on disk; the editor only displays it as N spaces visually.

Conversely, I’ve never heard a logical argument for using spaces. I frequently hear the justification that some editors don’t expand tabs correctly, but I’ve never actually found one that exhibits that misbehavior.

Log in to Reply

purencool says: February 4, 2011 at 11:37 pm

Thanks I need these reminders. Using the Java convention is of great advantage to any development.

Log in to Reply

Leave a Response

You must be logged in to post a comment.

← Bangspace — Bangkok Hackerspace

Singapore and Its Red Dot RubyConf →

META

- Register
- Log in
- Entries RSS
- Comments RSS
- WordPress.org

Search

Archives

- November 2015
- November 2013
- August 2013
- January 2013
- December 2012
- November 2012
- September 2012
- July 2012
- January 2012
- June 2011
- May 2011
- March 2010
- July 2009
- March 2009
- February 2009
- November 2008

Recent Activity

Posts **Comments**

- The Case for Nested Methods in Ruby
- Ruby's inject/reduce and each_with_object
- in Your System Prompt
- Using Oracle in JRuby with Rails and Sequel
- Copying (RVM) Data Between Hosts Using ssh, scp, and netcat
- Building A Great Ruby Development Environment and Desktop with Linux Mint 13 "Maya" Mate
- Intro to Functional Programming in Ruby
- WordPress Administration with Ruby
- Stealth Conditionals in Ruby

- Hello, Nailgun; Goodbye, JVM Startup Delays

Archives

Categories **Tags** **Dates** **Authors**

- Uncategorized