ChiefSimplicityOfficer.com

Courtesy of Craig Jones
(714) 955-4025
info@chiefsimplicityofficer.com

Java Coding Standard Suggestions

Updated 21-Dec-2011

Key Principles

Coding Standards

- Coding standards are subject to discussion and reexamination at any time. Agreed-upon coding standards are necessary in an Agile shop, where there is collective code ownership and constant refactoring. So, the standards must be adopted voluntarily by the whole team.[BECK00 p. 61]

Human Audience

- 99% of the time, it's more important to write code so that another programmer can read and understand it, than to write code so the computer can understand it.
- When writing code, work hard to avoid making the reader perform mental mappings. A five-minute investment up front will pay off 10 times over down the road. [MARTIN09 p. 25]

D.R.Y. – Don't Repeat Yourself

- Duplicate code is bugaboo #1 in almost every book on the subject of clean code [MARTIN09 p.289] [HUNT00 p. 26] [FOWLER99 p. 76].
- Martin says, every time you see duplicate code it's a missed opportunity for abstraction.

Fix Broken Windows

- Bad code tempts the mess to grow. Any code you modify should end up cleaner than you found it. [MARTIN09 p. 8]
- If there is insufficient time to fix it properly, then "board it up" – stub it out, change it to display "Not Implemented," or do something else to show that you are on top of the situation [HUNT00 p. 5]

Don't Speculate

- Aka. "No Spec-Gen" (Speculative Generality) [FOWLER99 p.83]
- Aka. YAGNI ("You aren't gonna need it")
- Aka. "Do the simplest thing that could possibly work" [BECK00 p.103]
- And no speculative performance tuning!

Single Responsibility Principle

- aka. Orthogonality [HUNT00 p. 34]
- Use very small classes – every class should have one and only responsibility (i.e. one reason to change). [MARTIN09 p. 138.]
- Related to the High Cohesion/Low Coupling principle [MARTIN09 p.140]

The Law of Demeter

- "A module should not know about the innards of the *objects* it manipulates." Martin points out, however, that not all Java classes are truly "objects." Sometimes they are simple data structures (e.g. beans and DTOs), and the Law of Demeter does not apply then. [MARTIN09 p. 97] [HUNT00 p. 138]

Test-Driven-Development

- "Unit tests are the water that keeps the potter's clay pliable." ~ Llewellyn Falco
- **Law #1:** Write no production code without a failing unit test first; **Law #2:** Write just enough unit test to fail; **Law #3:** Only write enough production code to make the test pass. [MARTIN09 p. 122]

Principle of Least Surprise

- Don't leave behaviors unimplemented that another programmer could reasonably expect.

"One difference between a smart programmer and a professional programmer is that the professional programmer understands that ***clarity is king***. Professionals use their powers for good and write code that others can understand." ~ *Robert (Uncle Bob) Martin*

References

[BECK00] – Beck, Kent; "Extreme Programming Explained: Embrace Change," 2000, Addison Wesley

[FOWLER99] – Fowler, Martin; "Refactoring: Improving the Design of Existing Code," 1999, Addison Wesley

[HUNT00] – Hunt, Andrew & Thomas, Dave; "The Pragmatic Programmer: From Journeyman to Master," 2000, Addison Wesley

[MARTIN09] – Martin, Robert; "Clean Code: A Handbook of Agile Software Craftsmanship," 2009, Prentice Hall

Naming

- Use pronounceable, meaningful names [MARTIN09 p. 17] that are intention-revealing
- Use fully spelled-out words, not abbreviations. Rare exceptions are well-known business domain acronyms (e.g. APR for annual-percentage-rate, BOM for bill-of-materials), and well-known programming idioms (e.g. FIFO for first-in-first-out, Dir for directory, Exe for executable). If you gave your code to another programmer to read out loud, cold, could he do so without stumbling?
- Note: When camel-casing an acronym, the JavaBean specification says to treat it like a word (`newAprConforming`, not `newAPRConforming`).

Basic Naming Conventions

`javaCase`	Local variables, class fields, method names, arguments/parameters
`CamelCase`	Classes and Interfaces – Do not name interfaces with an "I" prefix.
`ALL_CAPS`	Constants (final static)
`getXxxx()` and `setXxxx()`	Getters start with "get", Setters start with "set"
`findXxx()`, `determineXxx()`, `addXxx()`	If it's not a getter or a setter, then use another word besides "get" or "set"
`isXxxx()`	Boolean getters start with "is" (not "getIs", and not "are" even if that would be proper English)
`e`	Caught exceptions are always just "e"

Note: There are other, well-defined standards not mentioned here. For example, no one ever argues about how to make up a Java package name.

Functions

- One method should "do one thing" (and, by the way, error handling is considered one thing) [MARTIN09 p. 35 & 46]
- Favor class fields over method arguments [MARTIN09 p. 40]
- No side-effects [MARTIN09 p. 44]
- Do not mix commands and queries in the same method [MARTIN09 p. 45]
- Never return null (use a Null Object instead, e.g. Map.EMPTY_MAP)
- Do not return error codes. Throw exceptions.

On Removing Duplication

- Exactly repeated code can usually be fixed with Extract Method, or Extract Class.
- Repeated switch-statement and if-then-else chains can usually be fixed using polymorphism.
- Similar algorithms, with subtle differences in details, can usually be addressed with the Template Method and Strategy patterns.

Formatting

- It's vital that the code is uniformly indented so that statements within the same scope are aligned. ("[At] Bell Labs … [our] findings suggested that consistent indentation style was one of the most statistically significant indicators of low bug density." ~ James Coplien) [MARTIN09 p. xxii].
- Other formatting conventions [e.g. MARTIN p. 75-90] are useful, but of much less concern.
- Use your IDE's auto-formatting tools early and often (with the default settings), but don't reformat code that is already formatted well enough.
- When committing format changes to version control, try hard to commit them separately from significant code changes.

Objects vs. Data Structures

Data Structures

- e.g. Beans and DTOs
- No business logic, just public fields (or getter/setter methods if it must be a JavaBean)
- Law of Demeter does not apply

True Objects

- Hide the implementation
- Law of Demeter applies (avoid feature-envy, an indication that the classes aren't small enough).

Documentation

The one and only purpose to write documentation is if it enhances communication.

Permanent Documentation (maintained)

- Public interface APIs

Temporary Documentation (throw-away worksheets, as needed)

- Use-cases
- UML diagrams
- User Stories ("a contract for further discussion")
- Web flow wireframes

Commenting

- Avoid comments. (Explain yourself in the code itself as much as possible, and avoid the need for comments in the first place.) Comments are hard to maintain when the code is constantly changing and evolving.
- Prefer unit tests to documentation examples. A good unit test that serves as "executable documentation" is the picture that's worth a 1000 words.
- Comments should amplify – don't just repeat what's obvious by reading the code

Good Comments [MARTIN09 p. 55]	Bad Comments [MARTIN09 p. 59]
• Copyrights and other required legalities • Explanations of intent, clarifications, amplifications, and other truly informative comments • Warnings of consequences • JavaDocs in pubic APIs • `TODO`	• Mumbling, redundancies, and stating the obvious • Misleading comments (e.g. as often results from copy & paste errors) • @author, change history, etc. – Let the version control system keep track • "End comments" (closing-brace comments) – indicates that the code is too complicated, so just break it up • Commented-out code – Let the version control system keep track • HTML within JavaDoc – Most people read the JavaDocs directly out of the source and the tags just get in the way • Comments in the wrong place, too much information, unobvious connections, etc. • Function headers, section separators, and other noise • JavaDocs in non-public code • FIXME & XXX – use these temporarily to remind yourself of things to be fixed, but work quickly to remove them

Unit Tests

- Test code should be even cleaner code than the code being tested
- Single concept per test (which often means one assert per test, but not always)
- Evolve a domain-specific language for the testing.
- Use the "given-when-then" pattern for the DSL where applicable (i.e. private methods with names that start with "`given`", "`when`", "`then`") . Note: This is a variation of the "build-operate-check" pattern.

Error Handling

- Throw exceptions (not return codes)
- Use unchecked exceptions (extend RuntimeException) – Using checked exceptions violates the Open/Closed Principle.
- Provide context with exceptions

Databases

Table Schema

- Do not overload fields. For example, a status field that indicates if a record has been soft-deleted should not be used to indicate any other kind of status. In fact, a field name of just "status" is unacceptable. It should be more specific, e.g. "record_status", "password_status", "registration_status".
- Tables always have a long integer, auto-incremented, PK that's the same name as the table with an _id suffix.
- All table and field names are lower case with underscores (casing matters in the Linux version of MySQL)

Groovy-Specific

Duck Typing (Def)

- Prefer explicit typing over duck typing (because your IDE can't help you otherwise), except…
- Closure fields are always "Def" (e.g. in a Grails controller)

Obsolete Conventions & Practices

The following specific standards are no longer considered worthy:

Old	New	Why (Not)
Keeping names short `usrAcct`	Spelled-out, English-like, meaningful names `newlyRegisteredUserAccount`	It's more important to save the reader's time than the writer's time. When names are spelled out, the code reads like English prose. The reader is not burdened with having to transform mental mappings in order to read the code.
Hungarian notation or other "encodings" included in element names `String strCount = intCount.toString();`	Intention-revealing names `String iconBadge = newItemCount.toString();`	Modern IDEs can tell you the type of a variable just by hovering over it, so use the variable names to show why the variable exists, rather than how it's composed.
`I, j, k`, and `count` as for-loop index names	Intention-revealing names `sourceIndex, lastUsedBufferPosition`	Also, nested loops are too complicated. Try extracting the inner loop to its own method.
Function parameters names that begin with "a" or "an" `consolidate(Account anAccount);`	Just the name (with an indication of why it's being passed in) `consolidate(Account accountToDrop);`	
Using @author comments		Just let the version control system keep track. (Note: See the SVN BLAME command in Subversion.)
Endline comments on variable declaration lines. `Account account; //  the account to be merged-in and then dropped`	Intention-revealing names `Account accountToMergeThenDrop;`	
"Noisy" comments such as pro-forma documentation (function headers, etc.)	It's always better to use descriptive names for classes, methods, fields, variables and parameters than to document them.	Documentation has a habit of getting misplaced and going out of date. It's way better to let the code speak for itself.

Comment-out old code that might still be useful	Delete all unused code	Let the version control system keep track and keep the code uncluttered
Fields should always be declared private (and automatically create getters and setters for them)	Only do that when the object is required to conform to JavaBean standards (because a framework requires it); otherwise just use public fields.	Rely on an automated refactoring tool to generate a getter and/or setter later if/when it's needed
Automatically writing code in interface/impl pairs, just in case you decide later that you need to swap in a second implementation.	Writing just a single class and relying on an automated refactoring tool to extract the interface later if/when it's needed.	
Lots of inheritance	Less inheritance and more aggregation/composition and delegation	
Super classes that can be instantiated.	Super classes are always abstract. Only the "leaf nodes" of an object hierarchy can be instantiated.	
Configuration data in external files (.xml, .properties)	Configuration data in java source (where the refactoring tools can find them)	