Coding and Development Standards

Introduction and Motivation

I use the term *Coding Standards* to refer to guidelines pertaining to the appearance of the source code, and the term *Development Standards* to refer to more general programming guidelines that your organization agrees to adhere to. In this sense, coding standards are "easy" given that there are numerous code-checking and formatting tools that can be applied to make adherence to coding standards a mechanical practice. Development Standards on the other hand, require some conscious discipline on the part of the developer and, although their validation can be partially automated, probably need some additional enforcement mechanisms such as pair programming, peer review, and code review. Often Development Standards consist of avoiding practices that have been proven over time to result in poor quality code.

Let us examine the pros and cons of having standards in an organization. The primary reason for **not** having coding standards is that is nearly impossible for everyone to agree on them. Over time developers have evolved their own personal programming style and their are loath to modify it. Consequently, an enforced development standard will displease some people to some extend.

Now let us explore some of the main benefits that can reaped from adhering to a standard that will make it adopting it worthwhile.

Quality

Code Quality is the most obvious benefit of development standards. To give a simple example, consider several nested loops conditionals. If there are no enclosing brackets it’s hard to tell where each one starts and ends leading to the possibility of a bug. Even if the developer who wrote it is adept at this sort of thing, it will not be as obvious to another developer who tries to understand or modify the code. As a less trivial example, enforcing unit-testing standards, guarantees that code will be well-tested and largely regression-free before it hits QA.

Readability

Code Readability is another benefit of Coding Standards. Large classes and methods with deeply nested loops are notoriously hard to understand and even harder to test. It is also much harder to understand a piece of code, as you leap from class to class, if each file has its own programming style.

Maintainability

Last but not least comes the necessity of maintaining the code over time. Software applications have surprisingly long lifetimes, usually surpassing the employment span of developers who work on it. Code that is hard to read and understand is harder to maintain and more bug-prone.

Java Coding Standards

It is a testament to the importance of Coding Standards in Software Development that the Java language comes with an extensive set of style recommendations called the *Java Programming Style Guidelines.* The current url is http://www.oracle.com/technetwork/java/codeconvtoc-136057.html. It can be attested that the large majority of Java programmers are familiar and adhere to these guidelines by the fact that they are present in almost all professional open source software packages. Most IDEs come with validation and formatting tools that adhere to Java standards. It is usually sufficient for an organization to adopt these guidelines as its **Coding Standard** and supplement it with Development Standards.

Here, I describe some of the most important Java style guidelines.

- Have a consistent naming convention for classes. The Java standard is camelCase starting with a capital letter. Class names should not contain special characters including “underscore”.
- Have a consistent convention for method names. The Java standard is camelCase starting with a lowercase letter and should also not contain special characters.
- Have a consistent convention for variable names. The Java standard is camelCase starting with a lowercase letter and no special characters. The naming convention does not distinguish between member and local variables, which was a nice-to-have in the era of black-and-white editors but unnecessary in the world of modern IDEs.
- Have a convention for names of constants. In Java a constant is a variable demarcated as both **static** and **final**, i.e. a global immutable variable. The Java convention for constants in all uppercase separated by underscores.
- Come up with meaningful names. Whether you are naming a class, method, or variable give it a name that clearly identifies its purpose. Typically class names indicate an actor, as in someone who performs an action. For example,

RateCalculator. Methods typically start with a verb describing the action the method is responsible for, such as calculateTax().

- Standardize indentation and maximum line length. By setting all development IDEs to the same settings for indentation, line length, number of spaces between statements, et cetera, the code acquires a uniform typeset that makes it easy to read.
- Use brackets for loops and conditionals. Regardless of personal aesthetic taste, bracketed code is universally easier to read than non-bracketed code.
- Write small methods and classes. When designing a class, decide what are the responsibilities for it as a standalone object. Anything outside the scope of its responsibilities belongs to another class. Try to have each method perform one concrete unit of work.
- Use generics. Specify the type when you use utilities with generics such as collections. This will preclude storing the wrong type of object in a collection resulting in a ClassCastException. It also makes the code easier to read and maintain.

Development Standards

The collection of development standards presented here are the most commonly advocated by experienced Java programmers, architects, and authors. Because a list of dos and don’ts seems authoritative and arbitrary I added a section that justifies the adoption of the standard.

Avoid duplicating code

Why: Duplicating code means using copy and paste on code that is more than one line long. Duplicating code hinders code evolution and makes bugs much harder to fix, because when you modify a piece of code its duplicates remain hidden and unaltered.

How: Instead of copying the code that you want to reuse, refactor it into a method. If the code has to be used by multiple classes, then you can move this method in a utility class or into a superclass.

Do not hide exceptions

Why: Hiding exceptions means catching them and either doing nothing with them or printing an uninformative message. This means that when the code runs the root cause of a problem will go undetected.

How: If a method does not know how to recover from an exception is should throw it. In general, low level code like DAOs and I/O should throw exceptions and high level code such as servlets should handle them and either recover or fail gracefully. The code must throw meaningful exceptions that allow the caller to determine what happened. It is a good idea to have a package-wide exception extended by more specific exceptions. A good example of this pattern is IOException and FileNotFoundException. IOException signals the fact that the exception occurred at the IO layer and FileNotFoundException identifies what went wrong. Never throw Exception, Throwable, or RuntimeException because they are not informative.

Use a logging framework instead of print statements

Why: When the code runs in a production-like environment, all the print statements become invisible since there is typically no console access. Print statements introduce a performance hit but provide no tracing and debugging information.

How: Use a logging tool instead such as log4j, java logging, or sl4j.

Inject Dependencies in the constructor

Why: Prefer to pass dependencies into the constructor rather with get/set methods. This practice guarantees that objects are correctly initialized and it also prevents circular dependencies. For example, if your object uses a DataSource to read data, passing the DataSource in the constructor guarantees that the object is going to be ready to read as soon as it is created.

How: This is normally very easy to do unless you have a circular dependency.

Avoid stateful static methods

Why: As an example, consider a class that has a static getContext() method and a static setContext() method. This means that the context can change from anywhere at anytime and getContext() may return a different object every time.

How: Guarantee that static method calls do not alter the state of the class. If a class is stateful, provide non-static access to it. In general, it is a good practice to make upper tier classes, such as services, completely stateless.

Update and Check in often

Why: When multiple people are working on the same code, merging can be tricky. Updating and checking in often makes merges much smaller and easier to handler. Another reason for checking in often is that it encourages developing in small, iterative steps, so that if the code reaches an impasse - as it happens at times - it is easy to revert a small increment of it rather than start from scratch.

How: Developing code in small increments is a talent acquired by practice and discipline. As a general guideline, keeping your methods and classes small in size will make it much easier to develop incremental code. Learning refactoring techniques is also essential to developing code in small increments.

Unit Test

Why: Writing and running unit tests does not only guarantee that new code behaves as desired but also that existing code will not misbehave.

How: Writing a unit test is just as easy as writing any test, including a main() method but it comes with the additional advantage that the test lives for ever and guarantees that the behavior of your code will not change unexpectedly.

Set a Test Coverage Threshold

Why: If you are going to refactor code a lot (which is almost always the case) it is not sufficient to have a few scattered unit tests. The totality of the code must be well-tested to provide the confidence to make drastic changes and know that you have not changed the code's behavior unexpectedly.

How: Use a coverage tool such as Cobertura or EMMA.

Tools for Monitoring Standards

There are several tools that can help developers adhere to coding standards, measure deviations from the standards, and report on code quality in general.

To begin with, standard IDEs such as Eclipse and NetBeans have formatting and style-checking tools and they come packaged with the Java conventions. *Checkstyle* is a tool that lets an architect or manager define any coding standards and reports violations. It comes with ant tasks and plugins for Eclipse and NetBeans. *PMD* is an excellent code quality tool that detects possible bugs, dead code, suboptimal code, code duplication, and more. It is also highly configurable like checkstyle. *Findbugs* is a tool similar to PMD. *Cobertura* and *Emma* are tools that measure unit test coverage. Finally, *Hudson* is an excellent continuous integration tool for Java with plugins for incorporating reports produced by all of the tools mentioned here.