Code Review Checklist

http://commondatastorage.googleapis.com/bluelotussoftware/documents/Code%20Review%20Checklist.docx

Documentation

- ☐ All methods are commented in clear language. If it is unclear to the reader, it is unclear to the user.
- ☐ All source code contains @author for all authors.
- ☐ @version should be included as required.
- ☐ All class, variable, and method modifiers should be examined for correctness.
- ☐ Describe behavior for known input corner-cases.
- ☐ Complex algorithms should be explained with references. For example, document the reference that identifies the equation, formula, or pattern. In all cases, examine the algorithm and determine if it can be simplified.
- ☐ Code that depends on non-obvious behavior in external frameworks is documented with reference to external documentation.
- ☐ Confirm that the code does not depend on a bug in an external framework which may be fixed later, and result in an error condition. If you find a bug in an external library, open an issue, and document it in the code as necessary.
- ☐ Units of measurement are documented for numeric values.
- ☐ Incomplete code is marked with //TODO or //FIXME markers.
- ☐ All public and private APIs are examined for updates.

Testing

- ☐ Unit tests are added for each code path, and behavior. This can be facilitated by tools like Sonar, and Cobertura.
- ☐ Unit tests **must** cover error conditions and invalid parameter cases.
- ☐ Unit tests for standard algorithms should be examined against the standard for expected results.
- ☐ Check for possible null pointers are always checked before use.
- ☐ Array indices are always checked to avoid `ArrayIndexOfBounds` exceptions.
- ☐ Do not write a new algorithm for code that is already implemented in an existing public framework API, and tested.
- ☐ Ensure that the code fixes the issue, or implements the requirement, and that the unit test confirms it. If the unit test confirms a fix for issue, add the issue number to the documentation.

Error Handling

- ☐ Invalid parameter values are handled properly early in methods (Fast Fail).

- ☐ `NullPointerException` conditions from method invocations are checked.
- ☐ Consider using a general error handler to handle known error conditions.
- ☐ An Error handler **must** clean up state and resources no matter where an error occurs.
- ☐ Avoid using `RuntimeException`, or sub-classes to avoid making code changes to implement correct error handling.
- ☐ Define and create custom Exception sub-classes to match your specific exception conditions. Document the exception in detail with example conditions so the developer understands the conditions for the exception.
- ☐ (JDK 7+) Use try-with-resources. (JDK < 7) check to make sure resources are closed.
- ☐ **Don't pass the buck!** Don't create classes which throw Exception rather than dealing with exception condition.
- ☐ **Don't swallow exceptions!** For example `catch (Exception ignored) {}`. It should at least log the exception.

Thread Safety

- ☐ Global (static) variables are protected by locks, or locking sub-routines.
- ☐ Objects accessed by multiple threads are accessed only through a lock, or synchronized methods.
- ☐ Locks must be acquired and released in the right order to prevent deadlocks, even in error handling code.

Performance

- ☐ Objects are duplicated only when necessary. If you must duplicate objects, consider implementing Clone and decide if deep cloning is necessary.
- ☐ No busy-wait loops instead of proper thread synchronization methods. For example, avoid `while(true){ ... sleep(10);...}`
- ☐ Avoid large objects in memory, or using String to hold large documents which should be handled with better tools. For example, don't read a large XML document into a String, or DOM.
- ☐ Do not leave debugging code in production code.
- ☐ Avoid `System.out.println();` statements in code, or wrap them in a Boolean condition statement like `if(DEBUG) {...}`
- ☐ "Optimization that makes code harder to read should only be implemented if a profiler or other tool has indicated that the routine stands to gain from optimization. These kinds of optimizations should be well documented and code that performs the same task should be preserved."
  — **UNKNOWN**.