**Code Review Checklist**

**Please note that this is the full list contains mandatory as well as optional checklist. Although it is preferred to check all the items listed in the checklist, you may decide to skip some/all the optional checks in certain cases. If you are in double, please talk to your manager on this.**

**Maintainability**

1. Does the code make sense?
   * Make an effort to understand what the code is supposed to do before performing a code review. This can also be a part of the code review.
   * Require the developer to comment as much as necessary to make the code readable.
   * During code review, it may be necessary for lead reviewer to make comments in the code, with help from original developer.
2. Does the code comply with the accepted Coding Conventions?
   * Indentation
   * Variable / method names
   * Bracket style
3. Does it reflect the hierarchy of the existing class structure? Does it identify global, common and external classes, methods etc?
4. Has a common set of routines been written instead of replicating code for these routines in various programs?
5. Is there any redundant code? (There must be no redundant code.)
6. Does the code comply with the accepted Comment Conventions?
   * All classes and methods should contain a descriptive JavaDoc comment.
   * All methods should contain brief comments describing unobvious code fragments.
   * All class files should contain a copyright header.
   * All class files should contain class comments, including author name.
   * All public methods should contain comments that specify input parameters.
   * All methods which are > 20 lines should contain a comment that specifies ex3ecution details and possible return values.
   * Complex algorithms should be thoroughly commented.
   * Comment all variables that are not self-describing.
   * Static variables should describe why they are declared static.
   * Code that has been optimized or modified to “work around” an issue should be thoroughly commented, so as to avoid confusion and re-introduction of bugs.
   * Code that has been “commented out” should be explained or removed.
   * Code that needs to be reworked should have a TODO comment and a clear explanation of what needs to be done.
   * When in doubt, comment.
   * When you've commented too much, keep commenting.
   * When your wrists hurt from commenting too much, take a break ... and then comment more.

**Error Handling**

1. Does the code comply with the accepted Exception Handling Conventions.
   * We need to expand our notion of Exception Handling Conventions.
   * Some method in the call stack needs to handle the exception, so that we don’t display that exception stacktrace to the end user.
2. Does the code make use of exception handling?
   * Exception handling should be consistent throughout the system.
3. Does the code simply catch exceptions and log them?
   * Code should handle exceptions, not just log them.
4. Does the code catch general exception (java.lang.Exception)?
   * Catching general exceptions is commonly regarded as “bad practice”.
5. Does the code correctly impose conditions for “expected” values?
   * For instance, if a method returns null, does the code check for null?
     + What should be our policy for detecting null references?
6. Are the error messages understandable? Are the error messages adequate?
7. Have all errors been trapped and handled?
8. Does the code test all error conditions of a method call?
   * Make sure all possible values are tested.
   * Make sure the JUnit test covers all possible values.

**Security**

1. Does the code appear to pose a security concern?
   * Passwords should not be stored in the code. In fact, we have adopted a policy in which we store passwords in runtime properties files.
   * Connect to other systems securely – i.e. use HTTPS instead of HTTP where possible.
   * SQL injections

**Thread Safeness**

1. Does the code practice thread safeness?
   * If objects can be accessed by multiple threads at one time, code altering global variables (static variables) should be enclosed using a synchronization mechanism (synchronized).
   * In general, controllers / servlets should not use static variables.
   * Use synchronization on the smallest unit of code possible. Using synchronization can cause a huge performance penalty, so you should limit its scope by synchronizing only the code that needs to be thread safe.
   * Write access to static variable should be synchronized, but not read access.
   * Even if servlets/controllers are thread-safe, multiple threads can access HttpSession attributes at the same time, so be careful when writing to the session.
   * Use the volatile keyword to warn that compiler that threads may change an instance or class variable – tells compiler not to cache values in register.
   * Release locks in the order they were obtained to avoid deadlock scenarios.
2. Does the code avoid deadlocks?
   * I’m not entirely sure how to detect a deadlock, but we need to make sure we acquire/release locks in a manner that does not cause contention between threads. For instance, if Thread A acquires Lock #1, then Lock #2, then Thread B should not acquire Lock #2, then Lock #1.
   * Avoid calling synchronized methods within synchronized methods.

**Database**

1. Do queries on tables enforce the use of indexes?
2. Are add/modify mode screen attributes correctly set and reset?
3. Has error status been checked after each SQL statement?
4. Has locking been performed prior to updates?
5. Have the following conditions been checked in expressions?
   1. Rounding off  (if required)
   2. Possibility of division by zero

**Resource Leaks**

1. Have the following checks been performed?
   * checks for empty resource
   * checks for I-O error
2. Does the code release resources?
   * Close files, database connections, HTTP connections, etc.
3. Does the code release resources more than once?
   * This will sometimes cause an exception to be thrown.
4. Does the code use the most efficient class when dealing with certain resources?
   * For instance, buffered input / output classes.

**Control Structures**

1. Are all the branch conditions correct for loops?
2. Does a loop always terminate?
3. Is the condition for terminating a loop correct?
4. Have the divisors been tested for zero (where applicable)?
5. Can statements placed in the loop be placed outside the loop?
6. Does the code make use of infinite loops?
   * If so, please be sure that the end condition CAN and WILL be met.
7. Does the loop iterate the correct number of times?
   * Check initialization and end condition to make sure that the loop will be executed the correct number of times.
8. Has the portion of the code that the thread of execution never reaches been identified?
9. Do we Have too many nested ‘IF’ statements been used? (This should be avoided.)

**Reusability**

1. Is the code as generalized/abstracted as it could be?
2. Is the code a candidate for reusability?
   * If you see the same code being written more than once (or if you have copied-and-pasted code from another class), then this code is a candidate.

**Performance**

1. Will the requirements of execution time (if applicable) be met?
2. Is there a more efficient alternative?

**Methods/Functions**

1. TBD

**Java Primitives**

1. Are all the array indexes within bounds?
2. Are all the array indexes correctly initialized?

**Usability-related Issues**

1. Are information messages provided during processes that take a long time? (This is optional.)