Java Code Review Checklist

I consider the following points a MUST for any Java developer for when he/she is reviewing some Java code changes.

Checklist for developers

Code Quality

- Basic Checks (before)
  - The code compiles
  - Old unit tests pass
- The code was tested
  - The code was developer-tested
  - The new code must be covered by unit tests
  - Any refactoring must be covered by unit tests
  - At least 80% coverage for the code changes
  - Unit tests should be compliant to the standards.
- Clean Code
  - README file
  - GitIgnore file
  - Naming Conventions (classes, constants, variables, methods (void vs return), etc)
  - No hard-coded variables
  - Make sure it handles constants efficiently
  - Proper comment in code whenever required
  - Check for proper clean Up
    - Remove Console print statements
    - Remove Unnecessary comments
    - Use @deprecated on method/variable names that are not meant for future use
  - Handle strings appropriately (use StringBuilder or StringBuffer)
  - Optimize to use switch-case over multiple If-Else statements
  - Make sure variables don’t cause memory leaks
  - Indentation
  - Replace imperative code with lambdas and streams
  - No spelling mistakes
  - The code does what it says it does
  - The code is easy to read (**Readability**) && (**Understandability**)
  - Avoid duplicate code
  - Override hashCode when overriding equals
  - Reuse of existing code
  - Optimize Imports
  - Code dynamic
  - Use equals over ==
  - Avoid finalizers

    - Use enums instead of int constants
    - Return empty arrays or collections, not nulls
    - Always override toString
    - Check if the code could be replaced by calling functions in other libraries/components
- Best Practices
    - OOP Principles are correctly used
        - Abstraction
        - Polymorphism
        - Inheritance
        - Encapsulation
    - Code to interfaces
    - The code follows the SOLID PRINCIPLES
        - Single Responsibility Principle : A class should have one and only one responsibility. If class is performing more than one task, it leads to confusion.
        - Open & Close Principle : The developers should focus more on extending the software entities rather than modifying them.
        - Liskov Substitution Principle : It should be possible to substitute the derived class with base class.
        - Interface Segregation Principle : It's same as Single Responsibility Principle but applicable to interfaces. Each interface should be responsible for a specific task and should not have methods which he/she doesn't need.
        - Dependency Inversion Principle : Depend upon Abstractions- but not on concretions. This means that each module should be separated from other using an abstract layer which binds them together.
    - Design Patterns
        - Recommend a design pattern when you fill that a pattern could fit there
- Exception Handling

Branching Strategy

- Two Branch - master / develop
- Any feature branch uses develop branch and can be merged
- Fork a release branch off the develop branch where no further feature can be merged
- Once release branch is tested it can merged to master and develop branch
- Similarly hotfix branch can be created from master and can be merged to develop and master

Application Structure

- Ensure that your changes are correctly written on layers and it respects the Spring Boo App Structure
- Do not mixt up a Rest Service with a Web App (like Spring Rest with Spring MVC)

Model

- Understand the meaning of each model you defined and treat it like the most appropiate terminology for it: DTO (form for MVC), entity, Value Object, Java Bean
- Place it in the correct package

Rest API

- Ensure your Rest API respects the REST maturity levels
- Error Codes
    - Ensure that in case of any error or exception the API replies with the standard error codes (defined at the system level)
    - Do not write Rest API which have different directions for receiving requestions from
        - Example: having a service which facilitates communication with an external 3rd party, do not expose the same service for the 3rd party to request back, for this use a different service (e.g. a notification service)
        - You can get into a confusion when you have to handle differently the same exception in the service, but it comes from different directions, you have 2 clients and cannot reply with the same error codes (one should be internally and the other one specific to the 3rd party)
- DTO
    - Use DTO pattern for passing data between controller and service layer (at least when using **sensitive** or **aggregated** data)

MVC

- Ensure that your Model-View-Controller parts of the Web Application are compliant to the Best Practices
    - JSP don't have business logic (no Java code, only JSTL, Spring or Thymeleaf tags)
    - Model doesn't have business logic
    - Controller delegates the requests to the business (service) layer, it doesn't have logic
    - Use DTO (classes ending with Form in case of MVC) pattern for passing data between controller and service layer (at least when using **sensitive** or **aggregated** data)

Performance

- Release resources (HTTP connections, DB connections, Files, any I/O streams)
- Avoid excessive synchronization
- Keep Synchronized Sections Small
- Beware the performance of string concatenation
- Avoid creating unnecessary objects

Security

- Make class final if not being used for inheritance
- Avoid duplication of code
- Restrict privileges: Application to run with the least privilege mode required for functioning
- Minimize the accessibility of classes and members
- Document security related information
- Input into a system should be checked for valid data size and range
- Avoid excessive logs for unusual behavior
- Release resources (Streams, Connections, etc) in all cases
- Purge sensitive information from exceptions

- Do not log highly sensitive information
- Consider purging highly sensitive from memory after use
- Avoid dynamic SQL, use prepared statement
- Limit the accessibility of packages,classes, interfaces, methods, and fields
- Limit the extensibility of classes and methods (by making it final)
- Validate inputs (for valid data, size, range, boundary conditions, etc)
- Validate output from untrusted objects as input
- Define wrappers around native methods
- Make public static fields final
- Avoid serialization for security-sensitive classes
- Avoid exposing constructors of sensitive classes
- Guard sensitive data during serialization
- Be careful caching results of potentially privileged operations

Thead-safety

- Ensure your code is not a candidate for race conditions and deadlocks

Logging

- Logging for different level should be configurable.
- Log every transactions or the ones that require logging.
- Use appropriate log level corresponding to messages. For Ex: ERROR for exception.
- Always log execution time of method to check performance.

Alerting

Clean Code

Ensure that the code is clean as much as possible.

- Use Intention-Revealing Names
- Pick one word per concept
- Use Solution/Problem Domain Names
- Classes should be small!
- Functions should be small!
- Do one Thing
- Don't Repeat Yourself (Avoid Duplication)
- Explain yourself in code
- Make sure the code formatting is applied
- Use Exceptions rather than Return codes
- Don't return Null

Design Patterns

Recommend a design pattern where is the case

Naming Conventions

- Ensure variable, method, and class names convey the subject
- Use all lower cases for package names and use reversed Internet domain naming conventions
- Class names should start with Capitals
- Variable and method names should use CamelCase

App Structure

Check that the changes are correctly written on layers and it respects the Spring Boot App Structure Do not mix up a Rest Service with a Web App

Model

Understand the meaning of usage of each model and try to find where is the most appropriate to be placed

Exception Handling

Ensure that each exception is raised and handled correctly

- Beware of the NullPointerException
- Ensure that each exception is raised and handled correctly
- Ensure that you have a well-defined exception hierarchy
- Split the exceptions in two types: technical and business

Unit Testing

Coverage of minimum 80% Ensure that all corner cases are covered and it follows the unit testing standards

Rest API

Ensure that the API changes respect the REST maturity levels

Error Codes

Ensure that in case of a program error or exception, the API replies with the standard error codes Bad example is Hubject-Adapter

MVC

Ensure that the Model-View-Controller parts of the Web Application are compliant with the Best Practices