# Coding Guidelines | Java

https://wiki.sei.cmu.edu/confluence/display/java/SEI+CERT+Oracle+Coding+Standard+for+Java

**Clean Code**

|  |  |
| --- | --- |
|  | |
| Checklist Item | Category |
| Use Intention-Revealing Names | Meaningful Names |
| Pick one word per concept | Meaningful Names |
| Use Solution/Problem Domain Names | Meaningful Names |
| Classes should be small | Classes |
| Method should be small | Functions/Method |
| One Method Do one Thing | Functions/Method |
| Don't Repeat Yourself (Avoid Duplication) | Functions/Method |
| Explain yourself in code | Comments |
| Make sure the code formatting is applied | Functions/Method |
| Use Exceptions rather than Return codes | Functions/Method |
| Don't return Null | Functions/Method |
| Functions should not take too many input parameters. | Functions/Method |

**Security**

|  |  |
| --- | --- |
| Checklist Item | Category |
| Make class final if not being used for inheritance | Fundamentals |
| Avoid duplication of code | Fundamentals |
| Restrict privileges: Application to run with the least privilege mode required for functioning | Fundamentals |
| Minimize the accessibility of classes and members Declare the variables with the smallest possible scope.  Externalize configuration data in a .properties file. Sensitive information like password must be encrypted. | Fundamentals Fundamentals  Security |
| Document security related information | Fundamentals |
| Input into a system should be checked for valid data size and range to Avoid sql Injection | Denial of Service |
| Avoid excessive logs for unusual behavior | Denial of Service |
| Release resources (Streams, Connections, etc) in all cases after work done. | Denial of Service |
| Purge sensitive information from exceptions (exposing file path, internals of the system, configuration) | Confidential Information |
| Do not log highly sensitive information | Confidential Information |
| Consider purging highly sensitive data from memory after use | Confidential Information |
| Avoid dynamic SQL, use prepared statement and white list input validation like size,content,min,max | Injection Inclusion |
| Limit the accessibility of packages,classes, interfaces, methods, and fields | Accessibility Extensibility |
|  | |



|  |  |
| --- | --- |
| Validate inputs (for valid data, size, range, boundary conditions, etc) | Input Validation |
| Validate output from un trusted objects as input | Input Validation |
| Define wrappers around native methods (not declare a native method public) | Input Validation |
| Treat output from un trusted object as input | Mutability |
| Make public static fields final (to avoid caller changing the value) | Mutability |
| Avoid exposing constructors of sensitive classes | Object Construction |
| Avoid serialization for security-sensitive classes | Serialization Deserialization |
| Guard sensitive data during serialization | Serialization Deserialization |
| Be careful caching results of potentially privileged operations | Serialization Deserialization |

**Performance**



Checklist Item Category

Avoid excessive synchronization Concurrency

Keep Synchronized Sections Small Concurrency

Favor immutable objects. Security/Performance

Presence of long lived objects like ThreaLocal and static variables holding references to lots of short lived Performace/Memory objects. Performance

Reuse objects via flyweight design pattern.

Beware the performance of string concatenation use StringBuilder General Programming

Avoid creating unnecessary/Many objects Creating and Destroying

Objects

**General**

|  |  |
| --- | --- |
| Category | Checklist Item |
| checked exceptions for recoverable conditions and runtime exceptions for programming errors  Code to interface as opposed to implementation.  Avoid finalizers and properly override equals, hashCode, and toString methods. | Exceptions  Fundamentals Fundamentals |
| Favour the use of standard exceptions | Exceptions |
| Don't ignore exceptions | Exceptions |
| Check input parameters for validity and nulls | Methods |
| Return empty arrays or collections, not nulls | Methods |
| Minimize the accessibility of classes and members | Classes and Interfaces |
| In public classes, use accessor methods, not public fields | Classes and Interfaces |
| Minimize the scope of local variables | General Programming |
| Refer to objects by their interfaces | General Programming |
| Adhere to generally accepted naming conventions | General Programming |
| Avoid finalizers | Creating and Destroying Objects |
| Always override hashCode when you override equals | General Programming |

|  |  |
| --- | --- |
| Always override toString methods. | General Programming |
| Use enums instead of constants | Enums and Annotations |
| Use marker interfaces to define types | Enums and Annotations |
| Synchronize access to shared mutable data | Concurrency |
| Prefer executors to tasks and threads | Concurrency |
| Document thread safety | Concurrency |
| Valid JUnit test cases exist for business Services and data | Testing |

**Static Code Analysis**

|  |  |
| --- | --- |
| Category | Checklist Item |
| Check static code analyser sonar report for the classes added/modified | Static Code Analysis Soanr |

**Following are some of the Basic Java Standards Needs to be take care While doing Code:**

APIs should versioned.

Avoid creating unnecessary objects and always prefer to do Lazy Initialization

Object creation in Java is one of the most expensive operation in terms of memory utilization and performance impact. It is thus advisable to create or initialise an object only when it is required in the code.

Never make an instance fields of class public Always restrict the visiblity and use the accessor getter,setter whenever its is required. making a class field public can cause lot of issues in a program. For instance you may have a class called MyCalender. This class contains an array of String weekdays. You may have assume that this array will always contain 7 names of weekdays. But as this array is public, it may be accessed by anyone. Someone by mistake also may change the value and insert a bug!

best approach as many of you already know is to always make the field private and add a getter method to access the elements. Always try to minimize Mutability of a class

Making a class immutable is to make it unmodifiable. The information the class preserve will stay as it is through out the lifetime of the class. Immutable classes are simple, they are easy to manage. They are thread safe. They makes great building blocks for other objects. However creating immutable objects can hit performance of an app. So always choose wisely if you want your class to be immutable or not. Always try to make a small class with less fields immutable.

To make a class immutable you can define its all constructors private and then create a public static method to initialize and object and return it.

Try to prefer Interfaces instead of Abstract classes Always try to limit the scope of Local variable.

Local variables are great. But sometimes we may insert some bugs due to copy paste of old code. Minimizing the scope of a local variable makes code more readable, less error prone and also improves the maintainability of the code.

Always initialize a local variable upon its declaration. If not possible at least make the local instance assigned null value. Try to use standard library instead of writing your own from scratch

Writing code is fun. But “do not reinvent the wheel”. It is very advisable to use an existing standard library which is already tested, debugged and used by others. This not only improves the efficiency of programmer but also reduces chances of adding new bugs in your code. Also using a standard library makes code readable and maintainable. Wherever possible try to use Primitive types instead of Wrapper classes

Wrapper classes are great. But at same time they are slow. Primitive types are just values, whereas Wrapper classes are stores information about complete class.

Use Strings with utmost care.

Always carefully use Strings in your code. A simple concatenation of strings can reduce performance of program. For example if we concatenate strings using + operator in a for loop then everytime + is used, it creates a new String object. This will affect both memory usage and performance time.

Also whenever you want to instantiate a String object, never use its constructor but always instantiate it directly. Always return empty Collections and Arrays instead of null

Whenever your method is returning a collection element or an array, always make sure you return empty array/collection and not null. This will save a lot of if else testing for null elements.

Finally blocks should never have code that throws exception. Always make sure finally clause does not throw exception. If you

have some code in finally block that does throw exception, then log the exception properly and never let it come out :) While catching the exception either re throw it or log it so we will not lost the stack trace.

All Entities should override equals and hash code method.

In Spring Service layer should be transactional when ever we interacting with the DB .Use @Transaction annotation on the service layer never not on controller layer.

Use @Repository for your repositories.

Follow the First and basic principal of Rest Architecture and Guidelines i.e Use of proper Verb Get and Post For Every Get and Post Scenario in the project .For getting the resource post should not be used.

For Fileupload Use AWS S3 funcationality instead of dedicated server to maintain the scalability later.

Take care of SQL injection,CSRF attack,Cross Site Scripting while doing coding.Always while list the input do the validation for min,max,and length for server side whether its already there for UI.Escape/Encode the html character.

Do the proper logging Always initialise the logger with public static final.Never Use Sys-out use the logger debug option in devlopment for debugging and turn it off in production.Use the proper logging level like INFO,DEBUG,TRACE,ERROR for different scenario.