Java – Coding Guidelines

**Revision History**

|  |  |  |  |
| --- | --- | --- | --- |
| Rev. | Changes Made | Author | Date |
| 0.1 draft | Initial draft | Mihir Shelat |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

Table of Contents

[2 Introduction 4](#_Toc423259469)

[2.1 Purpose i.e. why do we need coding guidelines? 4](#_Toc423259470)

[2.2 Objective & Scope 4](#_Toc423259471)

[2.3 References 5](#_Toc423259472)

[3 Coding Standards 5](#_Toc423259473)

[3.1 Clean code 5](#_Toc423259474)

[3.1.1 Names – And opportunity to reveal intent 5](#_Toc423259475)

[5](#_Toc423259476)

[3.1.2 Meaningful Names 6](#_Toc423259477)

[3.1.3 Avoid Noise 6](#_Toc423259478)

[3.1.4 Code Structuring 7](#_Toc423259479)

[3.1.5 Some good practices 9](#_Toc423259480)

[4 Design Principles 12](#_Toc423259481)

[4.1 GRASP - General Responsibility Assignment Software Patterns 12](#_Toc423259482)

[4.1.1 Information Expert 13](#_Toc423259483)

[4.1.2 Creator 13](#_Toc423259484)

[4.1.3 Low Coupling 14](#_Toc423259485)

[4.1.4 High Cohesion 15](#_Toc423259486)

[4.2 SOLID Principles 17](#_Toc423259487)

[4.2.1 Single responsibility principle 17](#_Toc423259488)

[4.2.2 Open closed principle 18](#_Toc423259489)

[19](#_Toc423259490)

[4.2.3 Liskov Substitution Principle 19](#_Toc423259491)

[4.2.4 Interface Segregation Principle 21](#_Toc423259492)

[4.2.5 Dependency Injection or Inversion Principle 22](#_Toc423259493)

[4.3 Other Principles 23](#_Toc423259494)

[4.3.1 Programming for Interface not implementation 23](#_Toc423259495)

[4.3.2 Favor Composition over Inheritance 23](#_Toc423259496)

[4.3.3 Encapsulate what varies 23](#_Toc423259497)

[5 Best practices 24](#_Toc423259498)

[5.1 Java coding 24](#_Toc423259499)

[5.2 Java Performance 24](#_Toc423259500)

[5.3 Java Security 24](#_Toc423259501)

[6 Analyzers 25](#_Toc423259502)

[7 Checklist 26](#_Toc423259503)

[8 Effective Logging practices 27](#_Toc423259504)

# Introduction

## Purpose i.e. why do we need coding guidelines?

We have all seen situations where a new project starts whose objective is to develop from scratch an application in a leading-edge technology. Everything goes very fast; first, second, third release and then all of a sudden, the team's productivity & velocity starts to decrease. Fourth release is postponed for the third time, fixing something breaks something else.

Investing some efforts delivering good code quality code helps maintaining the code easily. Badly written code is difficult to maintain and bug-fixing, enhancement becomes error-prone and time consuming. Thus managing the source code quality is all about optimizing ROI – Return on Investment

As per Itay Maman - *A well-written program is a program where the cost of implementing a feature is constant throughout the program's lifetime*

In other words, a badly written program is a program where the cost of implementing a feature grows throughout time. Everyone must take care that code is not getting worse in subsequent releases. Some attention should be given to coverage of the code, following best practices and adhering to well-known standards.

Code which is not maintained rots & results in:

* Rigidity - hard to change because every change affects too many other parts of the system. A small changes causes the entire system to rebuild.
* Fragility - When you make a change, unexpected parts of the system break. In other words, changes to one module causes other unrelated modules to misbehave. Imagine a car system in which changing the radio station affects windows.
* Immobility - hard to reuse in another application because it cannot be disengaged from the current application caused by couplings and dependencies between different modules.

Whereas clean code is simple and direct. Clean code makes designers intent visible, which is full of abstractions and straight forward lines of controls.

## Objective & Scope

The objective is to define some guidelines & best practices to develop a clean code & maintain code quality throughout the lifecycle of project i.e. right from the beginning.

This document covers the following aspects of Java coding:

* Coding Standards
* Coding Principles
* Best practices
* Checklist
* Code Analyzers
* Logging practices

## References

# Coding Standards

Coding conventions are a set of guidelines for a specific programming language that recommend programming style, practices and methods for each aspect of a piece program written in this language. These conventions usually cover file organization, indentation, comments, declarations, statements, white space, naming conventions etc. These are guidelines for software structural quality & hence programmers are highly recommended to follow these guidelines to help improve the readability of their source code and make software maintenance easier. Reducing the cost of software maintenance is the most often cited reason for following coding conventions.

Both Oracle (i.e. Sun) & Google have come up with Java coding standards which are very similar & recommend similar coding styles.

Here is the Sun specific coding convention:

<http://www.oracle.com/technetwork/java/codeconventions-150003.pdf>

But considering Sun specific standards were last modified in year 2000, the following one which Google has suggested in 2014 is latest and relevant.

<https://google-styleguide.googlecode.com/svn/trunk/javaguide.html#s4.4-column-limit>

## Clean code

### Names – And opportunity to reveal intent

**public** List<**int**[]> getThem() {

List<**int**[]> list1 = **new** ArrayList<**int**[]>();

**for** (**int**[] x : theList)

**if** (x[0] == 4)

list1.add(x);

**return** list1;

}

Tell me what does it do?

### 

**public** List<Cell> getFlaggedCells() {

List<Cell> flaggedCells = **new** ArrayList<Cell>();

**for** (Cell cell : gameBoard)

**if** (cell.isFlagged())

flaggedCells.add(cell);

**return** flaggedCells;

}

And now?

What is the difference?

You express functionality which anyone understands in given context. Nothing should be left to imagination.

**public** **static** **void** copyChars(**char** a1[], **char** a2[]) {

**for** (**int** i = 0; i < a1.length; i++) {

a2[i] = a1[i];

}

}

**public** **static** **void** copyCharacters(**char** source[], **char** destination[]) {

**for** (**int** currentIndex = 0; currentIndex < source.length; currentIndex++) {

destination[currentIndex] = source[currentIndex];

}

}

Avoid mental mapping…

Don’t put reader to comprehend in their head what a1, a2 and i signify.

### Meaningful Names

This is incomplete information, Demands more...

* **getCustomer()**
* **retrieveCustomer()**
* **fetchCustomer()**

Which one to use?

Chose one & stick to it

**Int days;**

**Int daysSinceCreation;**

**Int daysSinceLastModification ;**

**Int durationInDays;**

Add context to make the name meaningful…

### Avoid Noise

The type is list so the name need not say it again ….

**List<Client> clientList;**

**List<Client> clients;**

**List<Client> regularClients;**

**List<Client> newClients;**

### Code Structuring

Make it read like a story…. Like a thriller movie, it should start with abstractions which become clearer as story unfolds until it gives you the Aha moment in the climax… Basically you need to refactor your code when you smell deeper problem such as:

* Duplicate code
* Long methods
* Big classes
* Big switch statements
* Long navigations (e.g. a.b().c().d())
* Too much checking for null objects
* Un-encapsulated fields / expressions.

#### Avoid Mixing Abstractions….

**public** **void** generateAggregateReportFor(

**final** List<StoryTestResults> storyResults,

**final** List<FeatureResults> featureResults) **throws** IOException {

LOGGER.info("Generating summary report for user stories to" + getOutputDirectory());

copyResourcesToOutputDirectory();

// create story report

Map<String, Object> storyContext = **new** HashMap<String, Object>();

storyContext.put("stories", storyResults);

storyContext.put("storyContext", "All stories");

addFormattersToContext(storyContext);

writeReportToOutputDirectory("stories.html",

mergeTemplate(STORIES\_TEMPLATE\_PATH).usingContext(storyContext));

// create feature report

Map<String, Object> featureContext = **new** HashMap<String, Object>();

addFormattersToContext(featureContext);

featureContext.put("features", featureResults);

writeReportToOutputDirectory(

"features.html",

mergeTemplate(FEATURES\_TEMPLATE\_PATH).usingContext(featureContext));

// generate home page for story and feature

generateReportHomePage(storyResults, featureResults);

// update history for feature

getTestHistory().updateData(featureResults);

generateHistoryReport();

}

Mixing abstractions …hard to understand… too many things going on

**private** **void** generateAggregateReportFor(

**final** List<StoryTestResults> storyResults,

**final** List<FeatureResults> featureResults) **throws** IOException {

copyResourcesToOutputDirectory();

generateStoriesReportFor(storyResults);

generateFeatureReportFor(featureResults);

generateReportHomePage(storyResults, featureResults);

updateHistoryFor(featureResults);

generateHistoryReport();

}

* Use extract method and move method techniques to create consistent abstractions…
* If you require more than few minutes to understand the intent of the method/class then, it probably needs improvement…

#### Encapsulate Complex Expressions…

**for** (TestStep currentStep : testSteps) {

**if** (!currentStep.isAGroup() && currentStep.getScreenshots() != **null**) {

**for** (RecordedScreenshot screenshot : currentStep.getScreenshots()) {

screenshots.add(**new** Screenshot(screenshot.getScreenshot().getName(),

currentStep.getDescription(), widthOf(screenshot.getScreenshot()),currentStep.getException()));

}

}

}

Explain yourself in code…

**if (employee.isEligibleForFullBenefits())**

**// Check to see if the employee is eligible for fullbenefits**

**if ((employee.flags & HOURLY\_FLAG) &&(employee.age > 65))**

When there is more complex conditional expressions (e.g. && / || etc) then abstract it for better readability…

**public boolean needsScreenshots() {**

**return (!isAGroup() && getScreenshotsAndHtmlSources() != null)**

**}**

**for (TestStep currentStep : testSteps) {**

**if (currentStep.needsScreenshots()) {**

What does this expression mean?

### Some good practices

#### Tell, Don’t Ask

* A method m of an object O may only invoke the methods of the following kinds of objects:
  + itself
  + m's parameters
  + Any objects created/instantiated within m
  + O's direct component objects
  + A global variable, accessible by O, in the scope of m
* Only talk to your immediate friends, don’t talk to strangers…
* In object oriented world, law can be stated simply as "use only one dot". That is, the code a.b.Method() breaks the law where a.Method() does not.

**walkDog(Dog dog){**

**dog.getLegs().move();**

**}**

**walkDog(Dog dog){**

**dog.walk();**

**}**

#### Don’t return or pass NULL

Null is a Billion dollar mistake – Avoid at all costs

Null was invented by Charles Antony Richard Hoare and he calls it his “billion dollar mistake”, it’s because failure to check have caused Null Pointers and in turn have causes huge losses commercially - <http://en.wikipedia.org/wiki/Tony_Hoare>

Java 8 has alternative idiom Optional<?> to handle situations of failure/null returns

//return Null if not found

**public** List<Employee> getEmployeesOnLeave(Date leaveDate) {

**return** selectEmployeeOnLeave(leaveDate);

}

//return Empty List if not found

**public** List<Employee> getEmployeesOnLeave(Date leaveDate) {

List<Employee> employeeOnLeave = selectEmployeeOnLeave(leaveDate);

If(employeeOnLeave==**null**) Collections.emptyList();

}

#### It’s good to be DRY (Don’t Repeat Yourself)

As name suggest DRY (don’t repeat yourself) means don’t write duplicate code, instead use abstraction to abstract common things in one place. If you use a hardcoded value more than one time consider making it public final constant, if you have block of code in more than two place consider making it a separate method.

**int** countItems(List<Item> items) {

**if**(items==**null** || items.isEmpty())

**return** -1;

**else** **if**(items!=**null** && !items.isEmpty())

....

}

Item selectItem(List<Item> items) {

**if**(items==**null** || items.isEmpty())

**return** Item.Blank;

**else** **if**(items!=**null** && !items.isEmpty())

....

}

**private** <T> isNotNullOrEmpty( **final** List<T> listUnderTest) {

**return** listUnderTest!=**null** && !listUnderTest.isEmpty();

}

**int** countItems(List<Item> items) {

**if**(isNotNullOrEmpty(items))

....

}

If you used common code to validate OrderID and SSN it doesn’t mean they are same or they will remain same in future. By using common code for two different functionality or thing you closely couple them forever and when your OrderID changes its format, your SSN validation code will break. So be aware of such coupling and just don’t combine anything which uses similar code but are not related.

#### Avoid Deep Nesting

**if** (cond1) {

System.***out***.println("cond1 is true");

**if** (cond2) {

System.***out***.println("cond1 and cond2 are both true");

**if** (cond3) {

System.***out***.println("cond1, cond2 and cond3 are all true");

}

}

}

**if** (cond1) {

System.***out***.println("Executing when cond1 is true");

}

**if** (cond1&& cond2) {

System.***out***.println("Executing when cond1 and cond2 are both true");

}

**if** (cond1 && cond2 && cond3) {

System.***out***.println("Executing when all cond1, cond2 & cond3 are true");

}

#### Move Method

**public** **class** Student {

**public** **boolean** isEnrolledFor(Course course) {

course.getEnrolledStudents().contains(**this**);

}

}

**public** **class** Course {

**private** List<Student> enrolledStudents;

**public** List<Student> getEnrolledStudents() {

**return** **this**.students();

}

}

**public** **class** Student{…}

**public** **class** Course{

**private** List<Student> enrolledStudents;

**public** List<Student> getEnrolledStudents(){

**return** **this**.enrolledStudents();

}

**public** **boolean** isEnrolledFor(Student student){

enrolledStudents.contains(student);

}

}

If a method on one class uses (or is used by) another class more than the class on which It’s defined, move it to the other class…

# Design Principles

While designing the application, some of the most obvious object oriented design questions we come across are:

* Which class has what responsibility?
* What will be the associations?
* Which object will create/contain which object?
* How to have minimum dependencies?
* I need to maximize cohesion and reduce coupling
* I need to increase the reuse etc. …

Many of us knows various design principles but have not put enough attention in following them while designing a solution. One should always strive for highly cohesive and loosely couple solution, code or design and looking open source code from Apache and Sun are good examples of Java design principles or how design principles should be used in Java coding. The following two are widely used principles to create a system that is easy to maintain and extend over time:

* **GRASP** i.e. General Responsibility Assignment Software Patterns (or Principles), consists of guidelines for assigning responsibility to classes and objects in object-oriented design.
* **SOLID** i.e. (Single responsibility, Open-closed, Liskov substitution, Interface segregation and Dependency inversion)

While these patterns/principles answer some software problem, and in almost every case these problems are common to almost every software development project. Please note:

* These are principles, not rules, laws or truth which must be followed under all conditions – not a panacea to all problems.
* These are advices, just like age old “An apple a day keeps doctor away” makes you healthy but does not actually keep the doctor away.
* So, at times, in pragmatic situation it may make perfect sense to overrule them but, you will know when you have and why …

## GRASP - General Responsibility Assignment Software Patterns

The different patterns and principles used in GRASP are: Controller, Creator, Indirection, Information Expert, High Cohesion, Low Coupling, Polymorphism, Protected Variations, and Pure Fabrication. All these patterns answer some software problem, and in almost every case these problems are common to almost every software development project.

* Here responsibility can be:
* Accomplished by a single object
* A group of object collaboratively accomplish a responsibility.
* GRASP helps us in deciding which responsibility should be assigned to which object/class.
* Identify the objects and responsibilities from the problem domain, and also identify how objects interact with each other.
* Define blue print for those objects – i.e. class with methods implementing those responsibilities.

### Information Expert

* **Problem:** How to assign responsibility a class?
* **Solution:** Assign a responsibility to the class that has the information necessary to fulfil the responsibility. The class which has all the information needed to perform operations, or in some cases they collaborate with others to fulfill their responsibilities. These responsibilities include methods, computed fields, and so on.

Cards

Game

Deck

Players

Has

Played by

Contains

**Question:** Which is class should have the logic of shuffling cards?

* Assume we need to get all the videos of a VideoStore

|  |
| --- |
| VideoStore |
| getAllVideos() |

|  |
| --- |
| Video |
| title |

* Since VideoStore knows about all the videos, we can assign this responsibility of giving all the videos can be assigned to VideoStore class.
* VideoStore is the information expert.

### Creator

* **Problem:** Who creates an Object? Or who should create a new instance of some class?
* **Solution:** Determine which class should create instances of a class based on the relationship between potential creator classes and the class to be instantiated

Assign class B responsibility of creating instance of class A if:

* + B aggregates A objects
  + B contains A objects
  + B records instances of A objects
  + B closely uses A objects
  + B has the initializing data for creating A objects

Where there is a choice, prefer B aggregates or contains A objects

Cards

Game

Deck

Players

Has

Played by

Contains

Deck contains Cards so, Deck should be creator of Card objects.

|  |
| --- |
| VideoStore |
| getAllVideos() |

|  |
| --- |
| Video |
| title |

* + Consider VideoStore and Video in that store.
  + VideoStore has an aggregation association with Video. I.e, VideoStore is the container and the Video is the contained object.
  + So, we can instantiate video object in VideoStore class

### Low Coupling

Coupling is a measure of how strongly one element is connected to, has knowledge of, or relies on other elements.

**Problem:** How to support low dependency, low change impact, increased reuse?

**Solution:** Assign responsibilities such that:

* lower dependency between the classes
* change in one class having lower impact on other classes
* higher reuse potential



Here class Rent knows about both VideoStore and Video objects. Rent is depending on both the classes.



VideoStore and Video class are coupled, and Rent is coupled with VideoStore. Thus providing low coupling.

### High Cohesion

High Cohesion is an evaluative pattern that attempts to keep objects appropriately focused, manageable and understandable. High cohesion is generally used in support of Low Coupling. High cohesion means that the responsibilities of a given element are strongly related and highly focused. Alternatively, low cohesion is a situation in which a given element has too many unrelated responsibilities. Elements with low cohesion often suffer from being hard to comprehend, hard to reuse, hard to maintain and averse to change

**Problem:** How to keep classes focused and manageable?

**Solution:** Assign responsibility so that cohesion remains high. Benefits of it are:

* Easily understandable and maintainable.
* Code reuse
* Low coupling

Here is the example of LOW Cohesion where all Student + database specific responsibilities are in the same class:



For High Cohesion, move database specific responsibilities to a DB class as show below.



## SOLID Principles

It's an acronym introduced by Michael Feathers for the "first five principles" of object-oriented programming and design named by Robert Martin. The principles (Single responsibility, Open-closed, Liskov substitution, Interface segregation and Dependency inversion), when applied together, intend to make it more likely that a programmer will create a system that is easy to maintain and extend over time.

### Single responsibility principle

The Single Responsibility Principle (SRP) states that there should never be more than one reason for a class to change. **One class should have one and only one responsibility**. This means that every class, or similar structure, in your code should have only one job to do.

Everything in the class should be related to that single purpose, i.e. be cohesive. It does not mean that your classes should only contain one method or property.

There can be a lot of members as long as they relate to the single responsibility. It may be that when the one reason to change occurs, multiple members of the class may need modification. It may also be that multiple classes will require updates.

**class** Employee {

**public** Money calculatePay() {...}

**public** **void** save() {...}

**public** String reportHours() {...}

}

Here we have pay 1) calculation logic with 2) database logic and 3) reporting logic all mixed up within one class. If you have multiple responsibilities combined into one class, it might be difficult to change one part without breaking others. Mixing responsibilities also makes the class harder to understand and harder to test, decreasing cohesion. The easiest way to fix this is to split the class into three different classes, with each having only one responsibility: database access, calculating pay and reporting, all separated.

So basically gather together the things that change for the same reasons. Separate those things that change for different reasons.

### Open closed principle

The Open-Closed Principle (OCP) states that classes (and even methods) should be open for extension but closed for modification. “Open to extension” means that you should design your classes so that new functionality can be added as new requirements are generated. “Closed for modification” means that once you have developed a class you should never modify it, except to correct bugs.

These two parts of the principle appear to be contradictory. However, if you correctly structure your classes and their dependencies, you can add functionality without editing existing source code.

Generally you achieve this by referring to abstractions for dependencies, such as interfaces or abstract classes, rather than using concrete classes. Functionality can be added by creating new classes that implement the interfaces.

Applying OCP to your projects limits the need to change source code once it has been written, tested and debugged. This reduces the risk of introducing new bugs to existing code, leading to more robust software.

Another side effect of the use of interfaces for dependencies is reduced coupling and increased flexibility.

**class** GraphicEditor {

**public** **void** drawShape(Shape s) {

**if** (s.m\_type == 1)

drawRectangle(s);

**else** **if** (s.m\_type == 2)

drawCircle(s);

}

**public** **void** drawCircle(Circle r) {....}

**public** **void** drawRectangle(Rectangle r) {....}

}

**class** Shape {

**int** m\_type;

}

**class** Rectangle **extends** Shape {

Rectangle() {

**super**.m\_type = 1;

}

}

**class** Circle **extends** Shape {

Circle() {

**super**.m\_type = 2;

}

}

What happens when you add a new shape?

* Impossible to add a new Shape without modifying GraphEditor
* Important to understand GraphEditor to add a new Shape
* Tight coupling between GraphEditor and Shape
* Difficult to test a specific Shape without involving GraphEditor

**if** (cond1) {

System.***out***.println("Executing when cond1 is true");

}

**if** (cond1&& cond2) {

System.***out***.println("Executing when cond1 and cond2 are both true");

}

**if** (cond1 && cond2 && cond3) {

System.***out***.println("Executing when all cond1, cond2 & cond3 are true");

}

### 

**class** GraphicEditor {

**public** **void** drawShape(Shape s) {

s.draw();

}

}

**class** Shape {

**abstract** **void** draw();

}

**class** Rectangle **extends** Shape {

**public** **void** draw() {

// draw the rectangle

}

}

A good example of this principle is a framework like struts or spring, you will see that you cannot change their core logic and request processing, But you can modify the desired application flow just by extending some classes and plugin them in configuration files.

### Liskov Substitution Principle

All the time we design a program module and we create some class hierarchies. Then we extend some classes creating some derived classes.

We must make sure that the new derived classes just extend without replacing the functionality of old classes. Otherwise the new classes can produce undesired effects when they are used in existing program modules.

Likov's Substitution Principle states that if a program module is using a Base class, then the reference to the Base class can be replaced with a Derived class without affecting the functionality of the program module.

All subclasses must, therefore, operate in the same manner as their base classes. The specific functionality of the subclass may be different but must conform to the expected behavior of the base class. To be a true behavioral subtype, the subclass must not only implement the base class’s methods and properties, but also conform to its implied behavior.

**public** **class** Product {

**public** string Name { get; set; }

**public** string Author { get; set; }

}

**public** **class** Book : Product {}

**public** **class** Movie : Product {}

If someone had a Product object (which was actually a Movie) and asked for the Author, what should it do (a Movie doesn’t have an Author)? – Must have seen UnsupportedOperationException, it’s a LSP violation. This principle is just an extension of the Open Close Principle and it means that we must make sure that new derived classes are extending the base classes without changing their behavior.

Here is one classic example violating Liskov’s Substitution principle. In the example 2 classes are used: Rectangle and Square. Let's assume that the Rectangle object is used somewhere in the application. We extend the application and add the Square class. The square class is returned by a factory pattern, based on some conditions and we don't know the exact what type of object will be returned. But we know it's a Rectangle. We get the rectangle object, set the width to 5 and height to 10 and get the area. For a rectangle with width 5 and height 10 the area should be 50. Instead the result will be 100

// Violation of Likov's Substitution Principle

**class** Rectangle {

**protected** **int** m\_width;

**protected** **int** m\_height;

**public** **void** setWidth(**int** width) {

m\_width = width;

}

**public** **void** setHeight(**int** height) {

m\_height = height;

}

**public** **int** getArea() {

**return** m\_width \* m\_height;

}

}

**class** Square **extends** Rectangle {

**public** **void** setWidth(**int** width) {

m\_width = width;

m\_height = width;

}

**public** **void** setHeight(**int** height) {

m\_width = height;

m\_height = height;

}

}

**class** LspTest {

**private** **static** Rectangle getNewRectangle() {

// it can be an object returned by some factory…

**return** **new** Square();

}

**public** **static** **void** main(String args[]) {

Rectangle r = LspTest.*getNewRectangle*();

r.setWidth(5);

r.setHeight(10);

// user knows that r it's a rectangle.

// It assumes that he's able to set the width and height as for the base class

System.***out***.println(r.getArea());

// now he's surprised to see that the area is 100 instead of 50.

}

}

### Interface Segregation Principle

It is applicable to interfaces as single responsibility principle holds to classes. It says: "Clients should not be forced to implement unnecessary methods which they will not use". Instead of one fat interface many small interfaces are preferred based on groups of methods, each one serving one submodule.

When we have non-cohesive interfaces, the ISP guides us to create multiple, smaller, cohesive interfaces. Smaller interfaces are easier to implement, improving flexibility and the possibility of reuse. As fewer classes share these interfaces, the number of changes that are required in response to an interface modification is lowered, which increases robustness.

interface Worker {

void work();

void eat();

}

ManWorker implements Worker {

void work() {…};

void eat() {30 min break;};

}

RobotWorker implements Worker {

void work() {…};

void eat() {

//Not Applicable for a RobotWorker

};

}

Bad example (polluted interface)

**interface** Workable {

**public** **void** work();

}

**interface** Feedable{

**public** **void** eat();

}

ManWorker **implements** Workable, Feedable {

**void** work() {...};

**void** eat() {...};

}

RobotWorker **implements** Workable {

**void** work() {...};

}

Solution: Split into two interfaces

Wherever you find that client of the interface doesn’t use/need all methods then create a specific interface and, the generic interface should extend the specific interface for backwards compatibility (if required).

### Dependency Injection or Inversion Principle

Don't ask for dependency it will be provided to you by framework. This has been very well implemented in Spring framework, beauty of this design principle is that any class which is injected by DI framework is easy to test with mock object and easier to maintain because object creation code is centralized in framework and client code is not littered with that.

The Dependency Inversion Principle (DIP) states that high-level modules should not depend upon low-level modules; they should depend on abstractions. Secondly, abstractions should not depend upon details; details should depend upon abstractions. The idea is that we isolate our class behind a boundary formed by the abstractions it depends on. If all the details behind those abstractions change, then our class is still safe. This helps keep coupling low and makes our design easier to change. DIP also allows us to test things in isolation, details like database are plugins to our system.

Below is an example which violates the Dependency Inversion Principle. We have the manager class which is a high level class, and the low level class called Worker. We need to add a new module to our application to model the changes in the company structure determined by the employment of new specialized workers. We created a new class SuperWorker for this.

Let's assume the Manager class is quite complex, containing very complex logic. And now we have to change it in order to introduce the new SuperWorker. Let's see the disadvantages:

* We have to change the Manager class (remember it is a complex one and this will involve time and effort to make the changes).
* Some of the current functionality from the manager class might be affected.
* Unit testing should be redone.

**class** Worker {

**public** **void** work() {

// ....working

}

}

**class** Manager {

Worker worker;

**public** **void** setWorker(Worker w) {

worker = w;

}

**public** **void** manage() {

worker.work();

}

}

**class** SuperWorker {

**public** **void** work() {

// .... working much more

}

}

Bad example of DI principle

All those problems could take a lot of time to be solved and they might induce new errors in the old functionality. The situation would be different if the application had been designed following the Dependency Inversion Principle. It means we design the manager class, an IWorker interface and the Worker class implementing the IWorker interface. When we need to add the SuperWorker class all we have to do is implement the IWorker interface for it. No additional changes in the existing classes.

**interface** IWorker {

**public** **void** work();

}

**class** Worker **implements** IWorker{

**public** **void** work() {

// ....working

}

}

**class** SuperWorker **implements** IWorker{

**public** **void** work() {

//.... working much more

}

}

**class** Manager {

IWorker worker;

**public** **void** setWorker(IWorker w) {

worker = w;

}

**public** **void** manage() {

worker.work();

}

}

As in this new design a new abstraction layer is added through the IWorker Interface, the problems from the bad code are solved:

* Manager class doesn't require changes when adding SuperWorkers.
* Minimized risk to affect old functionality present in Manager Class since we don't change it.
* No need to redo the unit testing for Manager Class.

|  |  |
| --- | --- |
| Principle | Tips |
| Single Responsibility Principle | A class should have one, and only one, reason to change. |
| Open Closed Principle | You should be able to extend a classes behavior, without modifying it |
| Liskov Substitution Principle | Derived classes must be substitutable for their base classes |
| Interface Segregation Principle | Make fine grained interfaces that are client specific |
| Dependency Inversion Principle | Depend on abstractions, not on concretions |

## Other Principles

### Programming for Interface not implementation

Always program for interface and not for implementation this will lead to flexible code which can work with any new implementation of interface. So use interface type on variables, return types of method or argument type of methods in Java.

### Favor Composition over Inheritance

Always favor composition over inheritance if possible. Some of you may argue this but I found that Composition is lot more flexible than Inheritance. Composition allows to change behavior of a class at runtime by setting property during runtime and by using Interfaces to compose a class we use polymorphism which provides flexibility of to replace with better implementation any time

### Encapsulate what varies

Only one thing is constant in software field and that is “Change”, so encapsulate the code you expect or suspect to be changed in future. Benefit of this OOPS Design principle is that It’s easy to test and maintain proper encapsulated code. If you are coding in Java then follow principle of making variable and methods private by default and increasing access step by step e.g. from private to protected and not public. Several of design pattern in Java uses Encapsulation, Factory design pattern is one example of Encapsulation which encapsulate object creation code and provides flexibility to introduce new product later with no impact on existing code.

# Best practices

## Java coding

### 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 initialize an object only when it is required in the code

**public class Countries {**

**private List countries;**

**public List getCountries() {**

**//initialize only when required**

**if(null == countries) {**

**countries = new ArrayList();**

**}**

**return countries;**

**}**

**}**

### Never make an instance fields of class public

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.

**private String[] weekdays = {"Sun", "Mon", "Tue", "Thu", "Fri", "Sat", "Sun"};**

**public String[] getWeekdays() {**

**return weekdays;**

**}**

But writing getter method does not exactly solve our problem. The array is still accessible. Best way to make it unmodifiable is to return a clone of array instead of array itself. Thus the getter method will be changed to.

**public String[] getWeekdays() {**

**return weekdays.clone();**

**}**

### 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 throughout 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.

**public class Employee {**

**private String firstName;**

**private String lastName;**

**//private default constructor**

**private Employee(String firstName, String lastName) {**

**this.firstName = firstName;**

**this.lastName = lastName;**

**}**

**public static Employee valueOf (String firstName, String lastName) {**

**return new Employee(firstName, lastName);**

**}**

**}**

### Try to prefer Interfaces instead of Abstract classes

First you cannot inherit multiple classes in Java but you can definitely implements multiple interfaces. It’s very easy to change the implementation of an existing class and add implementation of one more interface rather than changing full hierarchy of class.

Again if you are 100% sure what methods an interface will have, then only start coding that interface. As it is very difficult to add a new method in an existing interface without breaking the code that has already implemented it. On contrary a new method can be easily added in Abstract class without breaking existing functionality.

### 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.

Thus, declare a variable only when needed just before its use.

Always initialize a local variable upon its declaration. If not possible at least make the local instance assigned null value.

### 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 for loop then every time + 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. For example:

**//slow instantiation**

**String slow = new String("Yet another string object");**

**//fast instantiation**

**String fast = "Yet another string object";**

### Defensive copies are savior

Defensive copies are the clone objects created to avoid mutation of an object. For example in below code we have defined a Student class which has a private field birth date that is initialized when the object is constructed.

**public class Student {**

**private Date birthDate;**

**public Student(birthDate) {**

**this.birthDate = birthDate;**

**}**

**public Date getBirthDate() {**

**return this.birthDate;**

**}**

**}**

**//Now we may have some other code that uses the Student object**

**public static void main(String []arg) {**

**Date birthDate = new Date();**

**Student student = new Student(birthDate);**

**birthDate.setYear(2019);**

**System.out.println(student.getBirthDate());**

**}**

In above code we just created a Student object with some default birthdate. But then we changed the value of year of the birthdate. Thus when we print the birth date, its year was changed to 2019!

To avoid such cases, we can use Defensive copies mechanism. Change the constructor of Student class to following which ensure we have another copy of birthdate that we use in Student class

**public Student(birthDate) {**

**this.birthDate = new Date(birthDate);**

**}**

### Never let exception come out of finally block & Never throw “Exception”

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.

Also never throw java.lang.Exception directly. It defeats the purpose of using checked Exceptions. Also there is no useful information getting conveyed in caller method.

### Exception handling – Best practices

* Never swallow the exception in catch block
* Declare the specific checked exceptions that your method can throw
* Do not catch the Exception class rather catch specific sub classes
* Never catch Throwable class
* Always correctly wrap the exceptions in custom exceptions so that stack trace is not lost
* Either log the exception or throw it but never do the both
* Never throw any exception from finally block
* Always catch only those exceptions that you can actually handle
* Don't use printStackTrace() statement or similar methods
* Use finally blocks instead of catch blocks if you are not going to handle exception
* Remember "Throw early catch late" principle
* Always clean up after handling the exception
* Throw only relevant exception from a method
* Never use exceptions for flow control in your program
* Validate user input to catch adverse conditions very early in request processing
* Always include all information about an exception in single log message
* Pass all relevant information to exceptions to make them informative as much as possible
* Always terminate the thread which it is interrupted
* Use template methods for repeated try-catch
* Document all exceptions in your application in javadoc

### Other good practices

### Avoid using implementation types (i.e., HashSet). Instead use an interface (i.e. Set).

**public** **class** Bar {

**private** ArrayList list = **new** ArrayList(); // Use List instead

**public** HashSet getFoo() { // Use Set instead

**return** **new** HashSet();

}

}

**public** **class** Bar1 {

**private** List list = **new** ArrayList(); // Use List instead

**public** Set getFoo() { // Use Set instead

**return** **new** HashSet();

}

}

### Avoid Reassigning Parameters

Reassigning values to parameters is a questionable practice. Use a temporary local variable instead or use only when required.

**private void foo(String bar) {**

**bar = "something else";**

**}**

### Final Field Could Be Static

If a final field is assigned to a compile-time constant, it could be made static, thus saving overhead in each object at runtime.

**public static final int BAR = 42;**

### Position Literal First In Comparison

Position literals first in String comparisons - that way if the String is null you won't get a NullPointerException, it'll just return false.

**boolean bar(String x) {**

**return 2.equals("x");**

**}**

### Close resources

Ensure that resources (like Connection, Statement, and ResultSet objects) are always closed after use.

**public** **void** foo() {

Connection conn = pool.getConnection();

**try** {

// do stuff

} **catch** (SQLException ex) {

// handle exception

} **finally** {

conn.close();

}

}

### Never hardcode a path or drive name in code. Use relative path where possible.

### Use enum wherever required. Do not use numbers or strings to indicate discrete values

### Closing resources/stream

### Java Executor framework

### Make methods final by default

### Java 7 & 8 – Cool features

### Null Safe type

### Using NIO

### ……..

### ……..

### ……..

### ……..

### ……..

### ……..

### ……..

### ……..

## Java Performance

## Java Security

# Analyzers

# Checklist

# Effective Logging practices