1. It is almost impossible to maintain a software for its whole life by the original author.
2. Java Coding Best Practices and Standards help other developers (who are not the author of the code) to understand the code easily.
3. Code conventions and Standards also improve the readability of the software.
4. Allows developers to understand new code more quickly and thoroughly.
5. Sometimes source code is sent as a product, Code conventions and Standards help in making sure that the software is well packaged and clean as any other product.

**Use Proper Naming Conventions**

Proper naming convention for Java project.

Pre-decide names for every class, interfaces, methods and variables etc.

If other developers are also working with you on the same project, they should also follow it to maintain the uniformity.

A meaningful naming convention is extremely important as everything, from classes to interfaces are identified from their names in the code.

Do not assign random names just to satisfy the compiler, use meaningful and self-explanatory names so that it is readable and can be later understood by yourself, your teammates, quality assurance engineers and by staff who will be handling maintenance of the project.

**Class Members must be accessed privately**

Keep the accessibility of class fields as inaccessible as possible.

It is done to protect the fields.

To achieve that, private access modifier is the ideal choice.

This practice is recommended to maintain the encapsulation, one of the fundamental concepts of OOP.

Extremely basic concept but still do not properly followed

1. public class Teacher {

2. public String name;

3. public String subject;

4. }

 The encapsulation is compromised here as anyone can change these values like this,

1. Teacher T01 = new Teacher();

2. Teacher.name = "Sam";

3. Teacher.subject = “Science”;

 Using private access modifier with class members keeps the fields hidden preventing a user to change the data except for setter methods.

Here is another example using private access modifier:

1. public class Teacher {

2. private String name;

3. private String subject;

4.

5. public void setName(String name) {

6. this.name = name;

7. }

8. public void setSubject(String subject)

9. this.subject = subject;

10. }

11. }

**Never leave a Catch Blocks empty**

Preferred by elite developers to write a proper and meaningful message in the catch block while exception handling.

New developers often leave the catch block empty as initially they are the only ones working on a code but when the exception is caught by the empty catch block

If a program gets an exception, it does not show anything, making debugging harder and time-consuming.

**Use StringBuilder or StringBuffer for String Concatenation**

Using the “+” operator to join Strings together is a common practice in many programming languages including Java.

Concatenating numerous strings, the “+” operator proves to be inefficient as the Java compiler creates multiple intermediate String objects before creating the final concatenated string.

The java best practice, in that case, would be using “StringBuilder” or “StringBuffer”.

Built-in functions modify a String without creating intermediate String objects saving processing time and unnecessary memory usage.

For instance,

1. String sql = "Insert Into Users (name, age)";

2. sql += " values ('" + user.getName();

3. sql += "', '" + user.getage();

4. sql += "')";

 The above-mentioned code could be written using StringBuilder like this,

1. StringBuilder sqlSb = new StringBuilder("Insert Into Users (name, age)");

2. sqlSb.append(" values ('").append(user.getName());

3. sqlSb.append("', '").append(user.getage());

4. sqlSb.append("')");

5. String sqlSb = sqlSb.toString();

**Avoid Redundant Initializations**

Although it is very common practice, it is not encouraged to initialize member variables with the values: like 0, false and null. These values are already the default initialization values of member variables in Java. Therefore, a java best practice is to be aware of the default initialization values of member variables and avoid initializing the variables explicitly.

**Using enhanced for loops instead of for loops with counter**

“For” loop is used with a counter variable but a unique java best practice suggested by every top java developer is using the enhanced for loop instead of the old simple For loop.

Generally, it won’t make any difference to use either of them but in some cases, the counter variable used could be very error-prone.

The counter variable can incidentally get altered, it may get used later in the code or you may start the index from 1 instead of 0 which will result in disturbing the code at multiple points.

Consider the following code snippet:

1. String[] names = {"Sam", "Mike", "John"};

2. for (int i = 0; i < names.length; i++) {

3. method1(names[i]);

4. }

 Here variable “I” is used as a counter for a loop as well as the index for the array names. It can get problematic later in the code so We can avoid the potential problems by using an enhanced for loop like shown below:

1. For (String Name1 : names) {

2. Method1(Name1);

3. }

**Proper handling of Null Pointer Exceptions**

Null Pointer Exceptions are very common in Java.

This exception occurs in a result of an attempt to call a method on a Null Object Reference.

int noOfEmployees = office.listEmployees().count;

 This line is free of any error, but if either the object “office” or method” listEmployees()” is Null then the code will throw a null pointer exception.

Null pointer exceptions are inevitable but for its better handling, there are some java coding best practices to follow.

First, it is important to check Nulls prior execution so that they can be eliminated and alter your code to handle it well.

A corrected version of code is show below,

1. private int getListOfEmployees(File[] files) {

2. if (files == null)

3. throw new NullPointerException("File list cannot be null");

**Avoiding Memory leaks**

In Java, the developers do not have much control over memory management as Java manages the memory automatically.

Always releasing database connections after your querying is done,

Use of Finally block as often possible and Releasing instances stored in Static Tables are some java coding best practices you must adapt to prevent memory leakage.

**Proper Commenting**

As your code will be read by various people with varying knowledge of Java,

Proper comments should be used to give overviews of your code and provide additional information that cannot be perceived from the code itself.

Comments are supposed to describe the working of your code to be read by Quality assurance engineer, tester or maintenance staff who might not have Java skills

## Neglecting NullPointerException

If there is any object that can be a possible candidate of being null, we should always apply a null check. Can use the ‘Optional’ introduced in Java 8.

Sometimes  programmers either miss to apply null check or don’t apply it in a correct way.

While coding we should take an extra care for NullPointerException.

Point out the places where it may happen,

### **Avoid returning null values from empty Collections**

Empty collection is collection which hasn’t any elements.

Some developers assign null value for Collection which has no elements but this is a bad practice.

Instead of assigning null,

**Collections.EMPTY\_LIST, Collections.EMPTY\_SET** and **Collections.EMPTY\_MAP**etc**.**

**public static List<Location> getLocations() {**

**List<Location> listOfLocations = null; // (incorrect way)**

**List<Location> listOfLocations = Collections.EMPTY\_LIST; // (correct way) ​**

**........ // logic to get List of Locations**

**return listOfLocations;**

**}**

### **Use ‘Optional’ of Java 8**

The purpose of this call is to address the NullPointerException. Optional class can address some of these problems.

Apply a shorter solution, if we use Java 8 and later version of JDK. Here, we can use Optional type that can either be an empty object or a wrap of some value as below:

**Optional<String> optional = Optional.ofNullable(nullableString);**

**if(optional.isPresent()) {**

**System.out.println(optional.get());**

**}**

or even

**Optional<String> optional = Optional.ofNullable(nullableString);**

**optional.ifPresent(System.out::println);**

## Forgetting to free Resources after use

While reviewing the code, It has been observed that we forget to free the resources after use such as file, I/O streams, network connections etc.

Every time a program opens a resource, it should be an important practice to release them after use.

Similarly, we should have a special care if any exception were to be thrown during operations on such resources.

Java 7 has introduced a special statement try-with-resource that we can use proactively to manage it.

## Bypassing Exceptions

It is always one of the best practice to handle exceptions in the code where it is possible. Exceptions are thrown on a specific purpose, so in most cases we should address the issues causing these exceptions. If necessary, you can either rethrow it, show an error dialog to the user, or add a message to the log. At  least, we should explain the reason of an unhandled exception in order to let other developers find out the reason.

## Dealing With Concurrent Modification Exception

When we iterate over a collection using methods other than those provided by the iterator object, We can encounter a ConcurrentModificationException if modify the collection while iterating. For example, While iterating over a List of Invoices we want to remove the invoices which have amount greater than or equal to 100.

**List<Invoice> invoices = new ArrayList<>();**

**invoices.add(new Invoice("Inv1",98.5));**

**invoices.add(new Invoice("Inv2",118.75)); // Candidate for removal**

**invoices.add(new Invoice("Inv3",93.25));**

**for (Invoice invoice : invoices) {**

**if (invoice.getInvoiceAmount() >= 100.0) {**

**invoices.remove(invoice);**

**}**

**}**

If we run this code, we will encounter a ConcurrentModificationException, because the code modifies the List while iterating it.

Moreover, the same exception may arise if one of the multiple threads working with the same list and trying to modify the collection while others iterate over it. In that case, we should also think about the synchronization locks or some of the collections that deal with concurrent modification.

### Solution#1

We can collect all objects which are candidate for removal in a separate collection. Then remove them from another loop. But this method requires one additional collection for collecting the items.

**List<Invoice> invoicesForRemoval = new LinkedList<>();**

**for (Invoice invoice : invoices) {**

**if (invoice.getInvoiceAmount() >= 100.0) {**

**invoicesForRemoval.add(invoice);**

**}**

**}**

**for (Invoice invoice : invoicesForRemoval) {**

**invoices.remove(invoice);**

**}**

### Solution#2

We can use iterator() method of the collection to make it more concise as below:

**Iterator<Invoice> invoiceIterator = invoices.iterator();**

**while (invoiceIterator.hasNext()) {**

**Invoice invoice = invoiceIterator.next();**

**if (invoice.getInvoiceAmount() >= 100.0) {**

**invoiceIterator.remove();**

**}**

**}**

## Missing the ‘break’ Keyword in a Switch-Case Block

It is a very common mistake to miss break statement in switch-case block. For example, consider the below code

**int index = 0;**

**switch (index) {**

**case 0:**

**System.out.println("Sunday");**

**case 1:**

**System.out.println("Monday");**

**break;**

**case 2:**

**System.out.println("Tuesday");**

**break;**

**case 3:**

**System.out.println("Wednesday");**

**break;**

**case 4:**

**System.out.println("Thursday");**

**break;**

**case 5:**

**System.out.println("Friday");**

**break;**

**case 6:**

**System.out.println("Saturday");**

**break;**

**default:**

**System.out.println("Not a Day Of Week");**

**}**

If we missed to write a “break” in “case 0” in the code example above, the program will print “Sunday” followed by “Monday”, since the control flow inside here will go through the entire “switch” statement until it reaches a “break”.

## Overlooking Existing Libraries

Many a time, while doing code reviews It is observed that programmers ignore the existing libraries that can solve their purpose without writing an extra code to get a functionality. There are multiple reasons behind it. Sometimes it happens because of no knowledge of the existing library. A better way is to just check on the internet before the implementation if there is an existing library to get the functionality implemented.

For example, while applying try,-catch-finally in a program, we generally use finally to release resources that we have used in try-catch statement. We can avoid releasing resources, if we use try-with-resource, introduced in JDK 7. Similarly, we can implement many functionalities without writing an extra code, if we use the newer versions of JDK.

## Incorrect way to compare Objects

It has been observed from the code review that many developers incorrectly use the operator ‘==’ and equals() method. The ‘==’ operator compares if object references or equal or not. The equals() method compares the content of Objects. In most of the cases, we use the equals() method to compare two objects. Please have an extra care in using these two ways of comparing strings as we may get unexpected results if we don’t know the differences between them.

## Ignoring usage of Generics

We should always use the Generics to avoid any type of errors from raw types. Doing so, the compiler will detect any inconsistency right away and further minimize the chances to break the type safety.

## Common Standards/Conventions

Below are some common conventions that are important to discuss under the topic ‘Java Coding Best Practices and Standards’:

### Java Source Files

Each Java source file contains a single public class or interface. When private classes and interfaces are associated with a public class, you can put them in the same source file as the public class. The public class should be the first class or interface in the file.

#### Beginning Comments

All source files should begin with a c-style comment that lists the author(s), the version,  and also a brief description of the purpose of the program. For example:

/\*

\* Purpose Of the class

\*

\* Author

\*

\* Version info

\*/

#### Package and Import Statements

The first non-comment line of most Java source files is a package statement. After that, import statements can follow. For example:  
package java.util;  
import java.util.ArrayList;

#### Class and Interface Declarations

After Package declaration, the next turn is of Class and Interface Declarations. The first letter of the name of a class or interface should be in uppercase. If a class of interface has any annotation we need to apply the same before the Declaration of Class or Interface.

**@FunctionalInterface**

**interface MyInterface {**

**void m1();**

**}**

#### Class (static) Variables

The standard order of static variables is : first the public class variables, then the protected, and then the private.

#### Instance Variables

The standard order of instance variables is : first the public, then protected, and then private.

#### Constructors

In case of parameterized constructors, the constructors with lesser number of fields should come first.

#### Methods

The methods should be grouped by functionality rather than by scope or accessibility. For example, a private class method can be in between two public instance methods. The goal is to make reading and understanding the code easier.

### Comments

We should use Comments to give overviews of code and provide additional information that is not readily available in the code itself. Moreover, Comments should contain only information that is relevant to reading and understanding the program.

Java programs can have two kinds of comments: implementation comments and documentation comments. Implementation comments are delimited by /\*…   \*/, and //. Documentation comments (known as “doc comments”) are Java-only, and are delimited by /\*\*…\*/. Doc comments can be extracted to HTML files using the javadoc tool.

Implementation comments are for commenting out code or for comments about the particular implementation. Doc comments are to describe the specification of the code, from an implementation-free perspective, to be read by developers who might not necessarily have the source code at hand. The frequency of comments sometimes reflects poor quality of code. When you feel compulsory to add a comment, consider rewriting the code to make it clearer.

 ===============================================

<https://www.sonarqube.org/downloads/>

E:\Program Files\sonor\sonarqube-9.3.0.51899\bin\windows-x86-64

StartSonar.bat

<**build**>

<**pluginManagement**>

<**plugins**>

<**plugin**>

<**groupId**>org.sonarsource.scanner.maven</**groupId**>

<**artifactId**>sonar-maven-plugin</**artifactId**>

<**version**>3.4.0.905</**version**>

</**plugin**>

</**plugins**>

</**pluginManagement**>

</**build**>

mvn sonar:sonar -Dsonar.host.url=http://localhost:9000

-Dsonar.login=the-generated-token