CYBAGE
Delivering Value. Scientifically.

Java Coding Standard (JCS)

DOCUMENT NAME	Java Coding Standard (JCS)
VERSION NO.	1.2
RELEASE DATE	22-MAR-2021
CLASSIFICATION	Internal

Revision History

Ver. No.	Release Date	Authored / Modified by Date	Reviewed by Date	Approved by Date	Remarks / Change History
2.1	22-Mar-21				Migrated to new template and made cosmetic changes.

Table of Contents

1. Objective

The purpose of this document is to describe a collection of standards, conventions and guidelines for writing Java code that is easy to understand, to maintain and to enhance.

2. Introduction

2.1 Acknowledgement

This document is based on existing standard coding guideline described by *Oracle Corp*. (former *Sun Microsystems*). Existing standards from the industry are also used to provide guidelines for the coding. The reason behind each standard is also explained so that developers can understand the purpose to follow them.

These standards are based on proven software engineering practices that lead to improve development productivity, greater maintainability, and more scalability.

2.2 Why have Code Conventions

Code conventions are important to programmers for a number of reasons:

- 80% of the lifetime cost of a piece of software goes for maintenance.
- Hardly any software is maintained for its whole life by the original author.
- Code conventions improve the readability of the code software, allowing engineers to understand new code more quickly and thoroughly.
- If you deliver your source code as a product, you need to make sure it is as well packaged and clean as any other product you create.

For the code conventions to work properly, every developer responsible for writing the code must conform to the code conventions.

2.3 Target Audience

The audience for this document is all professional software developers who are involved in writing Java code and willing to enhance their productivity.

2.4 Prime Directive

A project requirement may vary from the standards mentioned in this document. Whenever there is a deviation from mentioned standards, projects should make sure to document it.

3. File Names

This section lists commonly used file suffixes.

3.1 File Suffixes

Java Software uses the following file suffixes:

Sr. No.	File Type	Suffix
1.	Java source	.java
2.	Java byte code	.class

4. File Organization

A file consists of different sections that should be separated by blank lines, and an optional comment should be added to identify each section.

Files longer than 2000 lines are cumbersome and should be avoided.

For example, of a properly formatted Java program, see "Java Source File Example".

4.1 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.

Java source files have the following order:

- Beginning comments (see "Beginning Comments")
- Package and Import statements
- Class and interface declarations (see "Class and Interface Declarations")

4.1.1 Beginning Comments

All source files should begin with a c-style comment that lists the class name, version information, date, and copyright notice:

```
/*
```

```
 * Classname
 *
 * Version information
 *
 * Date
 *
 * Copyright notice
 */
```

Additionally, this comment should be followed by a documentation comment.

```
/** @file <filename>
* Brief description of contents of file.
*
* Long description
*
* @date <date of creation of file>
* @version <CVS $ Header $ field>
*/
```

This will be used to generate automatic documentation using doxygen.

4.1.2 Package and Import Statement

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

```
package java.awt;

import java.awt.peer.CanvasPeer;
```

***Note*:** The first component of a unique package name is always written in all-lowercase ASCII letters and should be one of the top-level domain names, currently com, edu, gov, mil, net, org, or one of the English two-letter codes identifying countries as specified in ISO Standard 3166, 1981.

4.1.3 Class and Interface Declarations

The following table describes the parts of a class or interface declaration, in the order that they should appear. See “Java Source File Example” on page 18, for an example including comments.

Sr. No.	Part of Class/Interface Declaration	Notes
1.	Class/interface documentation comment (/**...*/)	See "Documentation Comments" for information on what should be in this comment.
2.	class or interface statement	
3.	Class/interface implementation comment (/*...*/), if necessary	This comment should contain any class-wide or interface-wide information that is not appropriate for the class/interface documentation comment.
4.	Class (static) variables	First the public class variables, then the protected, then package level (no access modifier), and then the private level.
5.	Instance variables	First public, then protected, then package level (no access modifier), and then private.
6.	Constructors	These methods should be grouped by functionality rather than by scope or accessibility. For example, a private class method can be in between the two public instance methods. The goal is to make reading and understanding the code easier.
7.	Methods	These methods should be grouped by functionality rather than by scope or accessibility. For example, a private class method can be in between the two public instance methods. The goal is to make reading and understanding the code easier.

5. Indentation

Four spaces should be used as the unit of indentation. The exact construction of the indentation (spaces vs. tabs) is unspecified. Tabs must be set exactly every 8 spaces (not 4).

5.1 Line Length

Avoid lines longer than 80 characters since they are not handled well by many terminals and tools.

Note: *Examples for use in documentation should have a shorter line length—generally, not more than 70 characters.*

5.2 Wrapping Lines

When an expression does not fit on a single line, break it according to these general principles:

- Break after a comma.
- Break before an operator.
- Prefer higher-level breaks to lower-level breaks.
- Align the new line with the beginning of the expression at the same level on the previous line.
- If the above rules lead to confusing code or to code that's squished up against the right margin, just indent 8 spaces instead.

Here are some examples of breaking method calls:

```
someMethod(longExpression1,longExpression2,longExpression3,
                                longExpression4,longExpression5);

var = someMethod1(longExpression1,
                  someMethod2(longExpression2,
                              longExpression3));
```

Following are two examples of breaking an arithmetic expression. The first is preferred, since the break occurs outside the parenthesized expression, which is at a higher level.

```
longName1 = longName2 * (longName3 + longName4 - longName5)
+ 4 * longname6;                                    // PREFER

longName1 = longName2 * (longName3 + longName4
- longName5) + 4 * longname6;                       // AVOID
```

Following are the two examples of indenting method declarations. The first is the conventional case. The second example would shift the second and third lines to the far right if it used conventional indentation, so instead it indents only 8 spaces.

```
//CONVENTIONAL INDENTATION
someMethod(int anArg, Object anotherArg, String yetAnotherArg,
                        Object andStillAnother) {
    ...
}

//INDENT 8 SPACES TO AVOID VERY DEEP INDENTS
private static synchronized horkingLongMethodName(int anArg,
```

```
            Object anotherArg, String yetAnotherArg,
            Object andStillAnother) {
        ...
}
```

Use the 8-space rule for line wrapping for if statements , since conventional (4 space) indentation does not give a clear view of the body. For example:

```
//DON'T USE THIS INDENTATION
if ((condition1 && condition2)
  || (condition3 && condition4)
  ||!(condition5 && condition6)) {                    //BAD WRAPS
  doSomethingAboutIt();                               //MAKE THIS LINE EASY TO MISS
}

//USE THIS INDENTATION INSTEAD
if ((condition1 && condition2)
        || (condition3 && condition4)
        ||!(condition5 && condition6)) {
   doSomethingAboutIt();
}//OR USE THIS
if ((condition1 && condition2) || (condition3 && condition4)
                        ||!(condition5 && condition6)) {
          doSomethingAboutIt();
}
```

Here are three acceptable ways to format ternary expressions:alpha = (aLongBooleanExpression) ? beta : gamma;

```
alpha = (aLongBooleanExpression) ? beta: gamma;alpha = (aLongBooleanExpression)? beta: gamma;
```

6. Comments

Java programs can have two kinds of comments: implementation comments and documentation comments. Implementation comments are those that are found in C++, which are delimited by /*...*/, and //. Documentation comments (known as "doc comments") are

Java-only and are delimited by /**...*/. Use the javadoc tool to extract the doc comments to HTML files.

Implementation comments are means for commenting code or for comments about a particular implementation. Doc comments are meant 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.

Comments should be used to give overview of the code and provide additional information that is not readily available in the code itself. Comments should contain only that information that is relevant to reading and understanding the program. For example, information about how a corresponding package is built or in which directory the package resides should not be included as a comment.

Discussion of nontrivial or non-obvious design decisions is appropriate but avoids duplicating information that is present in the code. In general, avoid any comments that are likely to get out of date as the code evolves.

Note: The frequency of comments sometimes reflects poor quality of code. When you feel compelled to add a comment, consider rewriting the code to make it clearer.

Comments should not be enclosed in large boxes drawn with asterisks or other characters.

Comments should never include special characters such as form-feed and backspace.

If you are removing the part of code, also remove the relevant comments from file.

Never keep the large chunk of commented code. It is often observed whenever logic changes, old code is commented out and new code is written.

6.1 Implementation Comments Format

Programs can have four styles of implementation comments: block, single-line, trailing, and end-of-line.

6.1.1 Block Comments

Block comments are used to provide descriptions of files, methods, data structures, and algorithms. Block comments are used at the beginning of each file and before each method. They can also be used in other places, such as within methods. Block comments inside
a function or method should be indented to the same level as the code they describe.

A blank line should precede a block comment to set it apart from the rest of the code.

```
/*
* Here is a block comment.
*/
```

Block comments can start with /*-, which is recognized by **indent**(1) as the beginning of a block comment that should not be reformatted. Example:

```
/*-
* Here is a block comment with some very special
* formatting that I want indent(1) to ignore.
*
* one
* two
* three
*/
```

Note: If you don't use **indent**(1), you don't have to use /*- in your code or make any other concessions to the possibility that someone else might run **indent**(1) on your code. See also, "Documentation Comments".

6.1.2 Single Line Comments

Short comments can appear on a single line indented to the level of the code that follows it. If a comment cannot be written in a single line, then it should follow the block comment format (see section 5.1.1). A blank line should precede a single-line comment. Here's an example of a single-line comment in Java code:

```
if (condition) {

    /* Handle the condition. */
    ...
}
```

6.1.3 Trailing Comments

Very short comments can appear on the same line as the code they describe, but there should be enough distance between the comments and the code. If more than one short comment appears in a chunk of code, they should all be indented to the same tab setting.

Here is an example of a trailing comment in Java code:

```
if (a == 2) {
    return TRUE;                    /* special case */
} else {
    return isPrime(a);              /* works only for odd a */
}
```

6.1.4 End of Line Comments

The // comment delimiter can be used to comment a complete line or a partial line. It should not be used on consecutive multiple lines for text comments; however, it can be used in consecutive multiple lines for commenting sections of code. Examples of all three styles follow:

```
if (foo > 1) {

    // Do a double-flip.
    ...
}
else{
```

```
        return false;           // Explain why here.
}

//if (bar > 1) {
//
//    // Do a triple-flip.
//    ...
//}
//else{
//        return false;
//}
```

6.1.5 Documentation Comments

Note: See, "Java Source File Example" on page 18 for examples of the comment format described here.

Comment tags are used to provide the additional information at the documentation level. These tags are to be used in below order:

@author (classes and interfaces only, required)
@version (classes and interfaces only, required)
@param (methods and constructors only)
@return (methods only)
@exception (@throws is a synonym added in Javadoc 1.2)
@deprecated

If desired, groups of the tags listed above, can be separated from the other tags by a blank line with a single asterisk.

Multiple **@author** tags should be listed in chronological order, with the creator of the class listed at the top.

Multiple **@param** tags should be listed in argument-declaration order. This makes it easier to visually match the list to the declaration.

Multiple **@throws** tags (also known as @exception) should be listed alphabetically by the exception names.

For further details, see "How to Write Doc Comments for Javadoc" which includes information on the doc comment tags (@return, @param, @see):

http://java.sun.com/products/jdk/javadoc/writingdoccomments.html

For further details about doc comments and javadoc, see the javadoc home page at:

http://java.sun.com/products/jdk/javadoc/

Doc comments describe Java classes, interfaces, constructors, methods, and fields. Each doc comment is set inside the comment delimiters /**...*/, with one comment per class, interface, or member. This comment should appear just before the declaration:

```
/**
* The Example class provides ...
*/
public class Example { ...
```

Notice that the top-level classes and interfaces are not indented, while their members are. The first line of doc comment (/**) for classes and interfaces is not indented; each subsequent doc comment lines have 1 space of indentation (to vertically align the asterisks). Members, including constructors, have 4 spaces for the first doc comment line and 5 spaces thereafter.

If you need to give information about a class, interface, variable, or method that is not appropriate for documentation, use an implementation block comment (see section 5.1.1) or single-line
(See section 5.1.2) comment immediately *after* the declaration. For example, details about the implementation of a class should go in such an implementation block comment *following* the class statement, not in the class doc comment.

Doc comments should not be positioned inside a method or constructor definition block, because Java associates documentation comments with the first declaration *after* the comment.

7. Declarations

7.1 Numbers per Line

One declaration per line is recommended since it encourages commenting. In other words,

```
int level;          // indentation level
int size;           // size of table
```

is preferred over

```
int level, size;
```

Do not put different types on the same line. Example:

```
int foo, fooarray[];                    //WRONG!
```

Note: The above examples use one space between the type and the identifier. Another acceptable alternative is to use tabs, e.g.:

```
int level;                              // indentation level
```

```
int size;                                   // size of table
Object currentEntry;                        // currently selected table entry
```

7.2 Initialization

Try to initialize local variables where they are declared. The only reason not to initialize a variable where it is declared is if the initial value depends on some computation occurring first.

7.3 Placement

Put declarations only at the beginning of blocks. (A block is any code surrounded by curly braces "{" and "}".) Do not wait to declare variables until their first use; it can confuse the unwary programmer and hamper code portability within the scope.

```
void myMethod() {
        int int1 = 0;                       // beginning of method block

        if (condition) {
                int int2 = 0;               // beginning of "if" block
                ...
        }
}
```

The one exception to the rule is indexes of for loops, which in Java can be declared in the for statement:

```
for (int i = 0; i < maxLoops; i++) { ... }
```

Avoid local declarations that hide declarations at higher levels. For example, do not declare the same variable name in an inner block:

```
int count;
...
myMethod() {
        if (condition) {
            int count;                      // AVOID!
             ...
        }
        ...
}
```

7.4 *Class* and *Interface* Declarations

When coding Java classes and interfaces, the following formatting rules should be followed:

- No space should be used between a method name and the parenthesis "(" starting its parameter list
- Open brace "{" appears at the end of the same line as the declaration statement.
- Closing brace "}" starts a line by itself indented to match its corresponding opening statement, except when it is a null statement the "}" should appear immediately after the "{"

```
class Sample extends Object {
        int ivar1;
        int ivar2;
        Sample(int i, int j) {
                ivar1 = i;
                ivar2 = j;
        }
        int emptyMethod() {}
        ...}
```

• Methods are separated by a blank line

• All methods should be preceded by a documentation comment describing the method, its arguments and return code(s).

```
/** The sample method.
*    This method calculates the sample thingy.
*    Some more description.
*
*  @param i The first argument.
*  @param j the second argument.
*
*  @return The sample value.
*/
int Sample(int i, int j) {
      return i+j;
}
```

8. Statements

8.1 Simple Statements

Each line should contain at most one statement. Example:

```
argv++;        // Correct
argc++;        // Correct
argv++; argc--;                              // AVOID!
```

8.2 Compound Statements

Compound statements are statements that contain lists of statements enclosed in braces "{statements}". See the following sections for examples.

- The enclosed statements should be indented one more level than the compound statement.
- The opening brace should be at the end of the line that begins the compound statement; the closing brace should begin a line and be indented at the beginning of the compound statement.
- Braces are used around all the statements; even single statements, when they are part of a control structure, such as an if-else or for statement. This makes it easier to add statements without accidentally introducing bugs due to missing braces.

8.3 Return Statements

A ***return*** statement with a value should not use parentheses unless they make the return value more obvious in some way. Example:

```
return;

return myDisk.size();

return (size ? size : defaultSize);
```

8.4 If, If else, If else – If else statements

The *if-else* class of statements should have the following form:

```
if ( condition) {
    statements;
}
if ( condition){
    statements;
} else          {
    statements;
}
if ( condition) {
    statements;
} else if ( condition) {
    statements;
} else {
```

```
    statements;
}
```

Note: if statements always use braces {}. Avoid the following error-prone form:

```
if ( condition)  //AVOID! THIS OMITS THE BRACES {}!
    statement;
```

8.5 For Statements

A *for* statement should have the following form:

```
for ( initialization; condition; update) {
   statements;
}
```

An empty for statement (one in which all the work is done in the initialization, condition, and update clauses) should have the following form:

```
for ( initialization; condition; update);
```

When using the comma operator in the initialization or update clause of a for statement, avoid the complexity of using more than three variables. If needed, use separate statements before the for loop (for the initialization clause) or at the end of the loop (for the update clause).

8.6 While Statements

A *while* statement should have the following form:

```
while ( condition) {
     statements;
}
```

An empty while statement should have the following form:

```
while ( condition);
```

8.7 Do while Statements

A *do-while* statement should have the following form:

```
do {
    statements;
```

8.8 } while (condition);Switch statements

A *switch* statement should have the following form:

```
switch ( condition) {
case ABC:
        statements;
        /* falls through */
case DEF:
        statements;
        break;
case XYZ:
        statements;
        break;
default:
        statements;
        break;
}
```

Every time a case falls through (does not include a break statement), add a comment where the break statement would normally be. This is shown in the preceding code example with the /* falls through */ comment.

Every switch statement should include a default case. The break in the default case is redundant, but it prevents a fall-through error if another case is added later.

8.9 Try Catch Statements

A *try-catch* statement should have the following format:

```
try {
     statements;
} catch (ExceptionClass e) {
     statements;
}
```

A *try-catch* statement may also be followed by finally,
which executes regardless of whether or not the try block has completed successfully.

```
try {
     statements;
} catch (ExceptionClass e) {
      statements;
} finally {
```

```
        statements;
}
```

9. Use of Java 1.5 features

9.1 Generics

Generics provides a way for you to communicate the type of a collection to the compiler, so that it can be checked.

Here is a simple example taken from the existing Collections tutorial:

```
// Code to removes 4-letter words from c. Elements must be strings
static void expurgate(Collection c) {
    for (Iterator i = c.iterator(); i.hasNext(); )
      if (((String) i.next()).length() == 4)
        i.remove();
}
```

Here is the same example modified to use generics:

```
// Removes the 4-letter words from c
static void expurgate(Collection<String> c) {
    for (Iterator<String> i = c.iterator(); i.hasNext(); )
      if (i.next().length() == 4)
        i.remove();
}
```

9.2 Enhanced *for* Loop

If using Java 1.5, make sure to use the Enhanced *For* Loop feature. This new language construct eliminates the drudgery and error-proneness of iterators and index variables when iterating over collections and arrays.

For example,

```
/* Conventional way */
void cancelAll(Collection<TimerTask> c) {
    for (Iterator<TimerTask> i = c.iterator(); i.hasNext(); )
        i.next().cancel();
}

/* Using Enhanced For Loop feature */
void cancelAll(Collection<TimerTask> c) {
    for (TimerTask t : c)
        t.cancel();
}
```

9.3 Autoboxing

This facility eliminates the drudgery of manual conversion between primitive types (such as int) and wrapper types (such as Integer).
For example:

```
Map<String, Integer> m = new TreeMap<String, Integer>();
    for (String word : args) {
       Integer freq = m.get(word);
       m.put(word, (freq == null ? 1 : freq + 1));
```

9.4 Typesafe *Enums*

This allows you to create enumerated types with arbitrary methods and fields. It provides all the benefits of the Typesafe Enum pattern

```
// int Enum Pattern - has severe problems!
public static final int SEASON_WINTER = 0;
public static final int SEASON_SPRING = 1;
public static final int SEASON_SUMMER = 2;
public static final int SEASON_FALL   = 3;

//Typesafe Enum
```

9.5 enum Season { WINTER, SPRING, SUMMER, FALL }Static Import

This facility lets you avoid qualifying static members with class names without the shortcomings of the "Constant Interface antipattern."

For example, use:

```
import static java.lang.Math.PI;
```

10. Use of Java 6 features

No language changes were introduced in Java SE 6. Hence, while working on Java 6, make sure to use language features from Java 5.

11. Use of Java 7 features

11.1 Type inference

JDK 1.7 introduces a new operator <>, known as diamond operator to making type inference available for constructors as well. Prior to Java 7, type inference is only available for methods

```
Map<String, List<Trade>> trades = new TreeMap<String, List<Trade>> ();
Map<String, List<Trade>> trades = new TreeMap <> ();
```

11.2 Using String in Switch statements

Prior to JDK 7, Switch statements work either with primitive types or enumerated types. In JDK 7, you can use a String object as the selector. For example,

```
String state = "NEW";
switch (day) {
   case "NEW": System.out.println("Order is in NEW state"); break;
   case "CANCELED": System.out.println("Order is Cancelled"); break;
   case "REPLACE": System.out.println("Order is replaced successfully"); break;
   case "FILLED": System.out.println("Order is filled"); break;
   default: System.out.println("Invalid");

}
```

11.3 Automatic Resource Management

In Java 7, you can use try-with-resource feature to automatically close resources, which implements AutoClosable and Closeable interface e.g., Streams, Files, Socket handles, database connections etc. JDK 7 introduces a try-with-resources statement, which ensures that each of the resources in try(resources) is closed at the end of the statement by calling close() method ofAutoClosable.

```
public static void main(String args[]) {
        try (FileInputStream fin = new FileInputStream("info.xml");
     BufferedReader br = new BufferedReader(new InputStreamReader(fin));) {
     if (br.ready()) {
        String line1 = br.readLine();
        System.out.println(line1);
   }
  } catch (FileNotFoundException ex) {
     System.out.println("Info.xml is not found"); }
  catch (IOException ex) {
     System.out.println("Can't read the file"); } }
```

11.4 Underscore in Numeric literals

In Java 7, you could insert underscore(s) '_' in between the digits in numeric literals (integral and floating-point literals) to improve readability. For example,

```
int billion = 1_000_000_000;  // 10^9
long creditCardNumber = 1234_4567_8901_2345L; //16-digit number
```

```
long ssn = 777_99_8888L;
double pi = 3.1415_9265;
float  pif = 3.14_15_92_65f;
```

11.5 Catching Multiple Exception Type in Single Catch Block

We can catch multiple exceptions in one catch block by using a '|' operator. This way, you do not have to write dozens of exception catches. However, if you have bunch of exceptions that belong to different types, then you could use "*multi multi-catch*" blocks too.

```
public void newMultiMultiCatch() {
    try{
        methodThatThrowsThreeExceptions();
     } catch(ExceptionOne e) {
    // log and deal with ExceptionOne
    } catch(ExceptionTwo | ExceptionThree e) {
    // log and deal with ExceptionTwo and ExceptionThree
    }
}
```

11.6 More Precise re-throwing of Exception

From JDK 7 onwards you can be more precise while declaring type of Exception in throws clause of any method. This leads to improved checking for re-thrown exceptions. You can be more precise about the exceptions being thrown from the method and you can handle them a lot better at client side, as shown in following example:

```
public void precise() throws ParseException, IOException {
    try {
        new FileInputStream("abc.txt").read();
        new SimpleDateFormat("ddMMyyyy").parse("12-03-2014");
    } catch (Exception ex) {
        System.out.println("Caught exception: " + ex.getMessage());
        throw ex;
    }
}
```

12. Use of Java 8 features

12.1 Lambda Expressions

This is a major feature in JDK 8, which helps developers pass the functionality as method argument or pass the code as data. The code can be written easily and meaningfully using Lambda expressions and streams. Lambda expression provides the simplest way to implement the interfaces with single method which will be more convenient to developers. **For example**, EventHandler interface has only one method called handle.

Prior to JDK 8, we need to write the code as below to override the handle method.

```
btn.setOnAction(new EventHandler<ActionEvent>() {
@Override
public void handle(ActionEvent event) {
    System.out.println("Hello World!");  }
});
```

The same can be written as below using lambda expression.

```
btn.setOnAction(
event -> System.out.println("Hello World!")
);
```

12.2 util.stream

This is a new package added in Java 8. This API is integrated into the Collections API. Using streams, the collections can be processed in sequential or parallel. Writing more than 5 lines of code to iterate a collection can be written in single line with stream and lambda expressions. **For example**, See the below code to sum the weight of all the widget having the color red in a collection of widget. Widgets is a collection having n number objects of Class Widget. Here stream and lambda expression made the work in single line instead of 5 lines of code.

```
int sum = widgets.stream().filter(w -> w.getColor() == RED)
.mapToInt(w -> w.getWeight()).sum();
```

12.3 Date-Time packages

Java.time is a new package added in jdk1.8 which comes with new api to resolve the drawbacks in the previous versions like no thread safe, poor design and difficult to handle time zone. All the classes in this new package are immutable and thread safe. This new package provides lot of date time related features, so there is no need to use any third-party date time API in most of the cases. For example: Let's take a simple scenario, you have an application which needs to force the user to change the password every 60 days. So, we need to find the number of days between password last changed date and the login date.

Prior to JDK 8, this was achieved using third party api's like below.

Using joda-time:

```
int passwordExpireDays = DAYS.daysBetween(passwordLastChangedDate, new Date()).getDays();
```

Using JDK 8:

```
long passwordExpireDays = ChronoUnit.DAYS.between(passwordLastChangedDate, new Date());
```

12.4 Parallelism

Parallelism means dividing the problem into sub problems and solving those problems simultaneously. Parallelism is achieved through Fork/join framework which was added in JDK 7. In JDK 8, parallelism is available in two places. One is Arrays sorting in parallel and second in executing streams in parallel. The Arrays class is enhanced with additional utility methods to perform parallel sorting on primitives and comparable objects.

For example,
Parallel sorting can be achieved through parallelSort() method in Arrays.
The collection classes are enhanced to provide the parallelstream like below:

```
int sum = widgets.parallelStream().filter(w -> w.getColor() == RED).mapToInt(w ->
w.getWeight()).sum();
```

13. White Space

13.1 Blank Lines

Blank lines improve readability of the code by setting off logically related sections of the code.

Two blank lines should always be used in the following circumstances:

- Between sections of a source file
- Between class and interface definitions

One blank line should always be used in the following circumstances:

- Between methods
- Between the local variables in a method and its first statement
- Before a block (see section 5.1.1) or single-line (see section 5.1.2) comment
- Between logical sections inside a method to improve readability

13.2 Blank Spaces

Blank spaces should be used in the following circumstances:

- A keyword followed by a parenthesis should be separated by a space.

```
Example:while (true) {
    ...
        }
```

Note that a blank space should not be used between a method name and its opening parenthesis. This helps to distinguish keywords from the method calls.

- A blank space should appear after commas in argument lists.
- All binary operators except . should be separated from their operands by spaces. Blank spaces should never separate unary operators such as unary minus, increment (“++”), and decrement (“--”) from their operands. Example:

```
a += c + d;
a = (a + b) / (c * d);

while (d++ = s++) {
      n++;
}
prints("size is " + foo + "\n");
```

-
-
- The expressions in a for statement should be separated by blank spaces.
- Example:

```
for (expr1; expr2; expr3)
```

- Casts should be followed by a blank space.
- Examples:

```
myMethod((byte) aNum, (Object) x);
myMethod((int) (cp + 5), ((int) (i + 3))
+ 1);
```

14. Naming Conventions

Naming conventions make programs more understandable by increasing their legibility. They can also give information about the function of the identifier—for example, whether it is a constant, package, or class—which can be helpful in understanding the code.

Sr. No.	Identifier Type	Rules for Naming	Examples
1.	Packages	The prefix of a unique package name is always written in all-lowercase ASCII letters and should be one of the top-level domain names. Currently com, edu, gov, mil, net, org, or one of the English two-letter codes identifying countries as specified in ISO Standard 3166, 1981.	com.sun.eng com.apple.quicktime.v2 edu.cmu.cs.bovik.cheese

Sr. No.	Identifier Type	Rules for Naming	Examples
		Subsequent components of the package name vary according to an organization's own internal naming conventions. Such conventions might specify that certain directory name components be division, department, project, machine, or login names.	
2.	Classes	Class names should be nouns, in mixed case with the first letter of each internal word capitalized. Try to keep your class names simple and descriptive. Use whole words; avoid acronyms and abbreviations (unless the abbreviation is much more widely used than the long form, such as URL or HTML).	class Raster; class ImageSprite;
3.	Interfaces	Interface names should be capitalized like class names.	interface RasterDelegate; interface Storing;

4.	Methods	Methods should be verbs, in mixed case with the first letter in lowercase, with the first letter of each internal word capitalized.	run();
			runFast();
			getBackground();

5.	Variables	Except for variables, all instance, class, and class constants are in mixed case with a lowercase first letter. Internal words start with capital letters. Variable names should not start with underscore _ or dollar sign $ characters, even though both are allowed. Variable names should be short yet meaningful. The choice of a variable name should be mnemonic— that is, designed to indicate to the casual observer the intent of its use. One-character variable names should be avoided, except for temporary "throwaway" variables. Common names for temporary variables are i, j, k, m, and n for integers; c, d, and e for characters.	int i; char c; float myWidth;
6.	Constants	Constants The names of variables declared class constants and of ANSI constants should be all uppercase with words separated by underscores	static final int MIN_WIDTH = 4;

Sr. No.	Identifier Type	Rules for Naming	Examples

		("_"). (ANSI constants should be avoided, for ease of debugging.)	static final int MAX_WIDTH = 999;
			
			static final int GET_THE_CPU = 1;

15. Programming Practices

Providing Access to Instance and Class Variables

Do not make any instance or class variable public without a good reason. Often, instance variables do not need to be explicitly set—often that happens as a side effect of method calls.

One example of appropriate public instance variables is the case where the class is essentially a data structure, with no behavior. In other words, if you would have used a struct instead of a class (if Java supported struct), then it is appropriate to make the class's instance variables public.

15.1 String concatenation

Avoid String concatenation "+=" : String concatenation is the textbook bad practice that illustrates the adverse performance impact of creating large numbers of temporary Java objects.

15.2 Referring to Class Variables and Methods

Avoid using an object to access a class (static) variable or method, use a class name instead.

For example:

```
classMethod();                                  //OK
AClass.classMethod();                          //OK
anObject.classMethod();                        //AVOID!
```

15.3 Constants

Numerical constants (literals) should not be coded directly, except for -1, 0, and 1, which can appear in a for loop as counter values.

15.4 Variables Assignments

Avoid assigning several variables to the same value in a single statement; it is hard to read.
Example:

```
fooBar.fChar = barFoo.lchar = 'c';                // AVOID!
```

Do not use the assignment operator in a place where it can be easily confused with the equality operator. Example:

```
if (c++ = d++) {                                  // AVOID! (Java disallows)
    ...
}
```

should be written as

```
if ((c++ = d++) != 0) {
     ...}
```

Do not use embedded assignments in an attempt to improve run-time performance. This is the job of the compiler.
Example:

```
d = (a = b + c) + r;                                                    // AVOID!
```

should be written as

```
a = b + c;
d = a + r;
```

15.5 Collections

15.5.1.1 Use interfaces to refer to collections objects instead of concrete class references.

E.g. List<String> names = new ArrayList<String>(); instead of ArrayList<String names> = new ArrayList<String>();

Always quantify generic collection references with concrete data types as shows above instead of using them as raw types. Always quantify generic interfaces with concrete data type while implementing or referring to.

15.5.1.2 Use for-each loop to iterate over collections unless element index value is specifically required.

15.5.1.3 Always use the most generic interface type applicable. E.g., use 'Set' instead of 'SortedSet' if 'Set' semantics are sufficient. This makes the code flexible.

15.5.1.4 Use a bounded type parameter when it will be later used in the code; use a bounded wildcard otherwise. Note that runtime erasure converts all quantified generic types to raw types; hence be careful when using 'instanceof' for quantified generic collections.

15.5.1.5 Do not mix raw and quantified collections in assignments, as parameters, or as return values.

15.5.1.6 If any such warnings still remain, ensure that they are benign. Relocate such warning generating code into specific methods and apply '@SupressWarnings("unchecked")' annotation to suppress such warnings.

15.5.1.7 Consider using 'checked' methods of 'Collections' class to enforce type safety on legacy raw collection objects. Ensure they cannot be accessed through their original, raw references.*

15.5.1.8 Consider using 'synchronized' methods of 'Collections' class to enforce synchronization on legacy raw collection objects when required. Ensure they cannot be accessed through their original, unsynchronized references.*

15.5.1.9 Consider using 'unmodifiable' methods of 'Collections' class to enforce non-modifiability on collection objects when required. Ensure they cannot be accessed through their original, modifiable references.*

15.5.1.10 Note that when accessing collection objects through their views, view modifying operations also modify the original collection objects.

15.5.1.11 Do not access elements of a LinkedList through index as such operations are quite expensive due to their requirement of linear traversal.

15.5.1.12 Avoid the use of Hashtable and use Hashmap instead. The key difference between the two is that access to Hashtable is synchronized on the table, while access to the HashMap isn't. The other advantage is that HashMap permits null values in it, while Hashtable doesn't.

15.5.1.13 Avoid the use of Vector and use ArrayList instead. Both Vector and ArrayList are considered very similar in that both of them represent an array that can be increased and where the elements can be accessed through an index.

15.5.1.14 Consider using TreeSet and TreeMap when elements/keys need to be sorted according to their natural order.

15.6 Miscellaneous Practices

15.6.1 Parentheses

It is generally a good idea to use parentheses liberally in expressions involving mixed operators to avoid operator precedence problems. Even if the operator precedence seems clear to you, it might not be to others—you should not assume that other programmers know precedence as well as you do.

```
if (a == b && c == d)              // AVOID!
if ((a == b) && (c == d))          // USE
```

15.6.2 Returning Values

Try to make the structure of your program match the intent. Example:

```
if ( booleanExpression) {
    return true;
} else {
    return false;
}
```

should instead be written as

```
return booleanExpression;
```

Similarly,

```
if (condition) {
   return x;
}

return y;
```

should be written as

```
return (condition ? x : y);
```

15.6.3 Expressions before '?' in the Conditional Operator

If an expression containing a binary operator appears before the ? in the ternary ?: operator, it should be parenthesized. Example:

```
(x >= 0) ? x : -x;
```

15.6.4 Special Comments

Use XXX in a comment to flag something that is bogus but works. Use FIXME to flag something that is bogus and broken.

15.6.5 Interfaces for Constants

It's possible to place widely used constants in an interface. If a class implements such an interface, then the class can refer to those constants without a qualifying class name. Interfaces are to expose the contract not to hold the implementation details or constants. Hence placing constants in an interface (constant interface pattern) is an anti-pattern and should be avoided. Below points will be helpful for your discretion on how to define constants.

1. If the constants need not to visible outside class. Define it in the class where it is being used.
2. If the constants are visible across the classes and can be grouped, then define them in enum.
3. If the constants are visible across the classes and can be grouped, then define them in a normal class.

15.6.6 Other Practices:

15.6.6.1 The lines of code in a Java class file should not be excessive. As a general guideline, anything past 1200 lines may be excessive.

15.6.6.2 The lines of code in a Java method should not be excessive. As a general guideline, anything past 100 lines may be excessive.

15.6.6.3 The number of characters in a line of code should not be excessive. As a general guideline, anything past 120 characters may be excessive.

15.6.6.4 Declare each variable on a new line.

15.6.6.5 Declare class variable as private and use getter and setter methods to access it outside of the class.

15.6.6.6 Do not use System.out.println – refer to “Logging Framework”.

15.6.6.7 Do not make fields public without good reason.

15.6.6.8 Import individual classes specifically - do not use * imports.

15.6.6.9 Use System.gc for explicit garbage collection in case of some memory intensive task, though java run time performs the garbage collection by itself.

15.6.6.10 After try and catch block use finally block to clean up or free the resources.

15.6.6.11 Use constants in place of hard coded values.

15.6.6.12 Avoid the use of properties (.properties extension) files. If properties are defined with a .properties file, they can become inconsistent, poorly documented and easily broken when the size and the complexity

of the content increases. XML configuration files are easier to organize, read and maintain. XML configuration files can also handle and maintain complex content in an easier way.

15.6.6.13 Watch out for use of "==" vs. "equals()" when comparing objects.

15.6.6.14 Assign null to object references that are no longer used.

15.6.6.15 Avoid use of deprecated methods (http://java.sun.com/j2se/1.5.0/docs/api/deprecated-list.html)

15.6.6.16 Use 'enum' data type instead of creating ad-hoc groups of constants.

15.6.6.17 Avoid passing many parameters to a method. If a method needs more than 3 parameters to be passed to it, consider refactoring or adding to instance variables of the class.

15.6.6.18 If a method's business logic throws an exception, specify it in its signature and write a separate method to handle it.

15.6.6.19 Use a ternary operator and avoid unnecessary if-else statement. If an expression containing a binary operator appears before the ? in the ternary ?: operator, it should be parenthesized. Example: (x >= 0) ? x : -x;

15.6.6.20 Avoid assigning several variables to the same value in a single statement; it is hard to read.

Example: fooBar.fChar = barFoo.lchar = 'c'; // AVOID!

15.6.6.21 Use checked exceptions to indicate problems that cannot be fixed at runtime.

15.6.6.22 Do not hard-code any platform specific values in code (e.g. file path separators). Use appropriate Java supplied constants instead.

15.6.6.23 Ensure that loops will always terminate.

15.6.6.24 Ensure existence of resources before accessing them.

15.6.6.25 Make all classes final unless specifically designed for subclassing.

15.6.6.26 Make all fields private unless designed to be accessed by subclasses. Never make them public.

15.6.6.27 Provide accessor methods to access state of an object.

15.6.6.28 Make all classes immutable unless designed for capturing mutable state.

15.6.6.29 Always use deep cloning when returning mutable objects of a class's private state.

15.6.6.30 Do not use finalizers as far as possible, write separate methods to clear resource e.g. when an object is done with using them, and document the fact that client code should call those methods at the end of its object usage.

15.6.6.31 When overriding 'equals' method, also override 'hashCode'.

15.6.6.32 When overriding 'clone', be aware that Java's default clone implementation creates a shallow copy. Override it correctly in case of a deep copy is needed. Consider using factory copy methods or copy

constructors instead of using clone. Ensure that the clone() method calls super.clone() otherwise may produce an object of the wrong class.

16. Code Examples

16.1 Java Source File Example

The following example shows how to format a Java source file containing a single public class. Interfaces are formatted similarly. For more information, see "Class and Interface Declarations" and "Documentation Comments"

```
/*
 * @(#)Blah.java                                1.82 99/03/18
 *
 * Copyright (c) 1994-1999 Sun Microsystems, Inc.
 * 901 San Antonio Road, Palo Alto, California, 94303, U.S.A.
 * All Rights Reserved.
 *
 * This software is the confidential and proprietary information of Sun
 * Microsystems, Inc. ("Confidential Information"). You shall not
 * disclose such Confidential Information and shall use it only in
 * accordance with the terms of the license agreement you entered into
 * with Sun.
 */


package java.blah;

import java.blah.blahdy.BlahBlah;
/**
 * Class description goes here.
 *
 * @version                                     1.82 18 Mar 1999
 * @author                                      Firstname Lastname
 */
public class Blah extends SomeClass {
/* A class implementation comment can go here. */

/** classVar1 documentation comment */
public static int classVar1;

/**
 * classVar2 documentation comment that happens to be
 * more than one line long
 */
```

```
private static Object classVar2;

/** instanceVar1 documentation comment */
public Object instanceVar1;

/** instanceVar2 documentation comment */
protected int instanceVar2;

/** instanceVar3 documentation comment */
private Object[] instanceVar3;

/**
 * ... constructor Blah documentation comment...
 */
public Blah() {
// ...implementation goes here...
                    }
/**
 * ... method doSomething documentation comment...
 */
public void doSomething() {
// ...implementation goes here...
                    }

/**
 * ...method doSomethingElse documentation comment...
 * @param someParam description
 */
public void doSomethingElse(Object someParam) {
// ...implementation goes here...
                    }
}
```