**Java Coding Standards**

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

### 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.

### The source file name consists of the case-sensitive name of the top-level class

### Source files are encoded in UTF-8.

## **Source file structure**

1. License or copyright information, if present

/\*

\* Classname

\*

\* Version information

\*

\* Date

\*

\* Copyright notice

\*/

1. Package statement
2. Import statements
   1. No wildcard imports
   2. Can do wildcard imports static imports
   3. Import statements are not line-wrapped
   4. All static imports in a single block.
   5. All non-static imports in a single block.
   6. If there are both static and non-static imports, a single blank line separates the two blocks. There are no other blank lines between import statements.
3. Exactly one top-level class, Each top-level class resides in a source file of its own.

## **Indentation**

## Line Length

* Oracle: 80 characters
* Google : 100 characters

### Wrapping Lines : When an expression will not fit on a single line, break it according to these general principles:

### Break after a comma.

* Break before an operator.

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

var = someMethod1(longExpression1,

someMethod2(longExpression2,

longExpression3));

* Prefer higher-level breaks to lower-level breaks.

longName1 = longName2 \* (longName3 + longName4 - longName5)

+ 4 \* longname6; // PREFER

longName1 = longName2 \* (longName3 + longName4

- longName5) + 4 \* longname6; // AVOID

* A line is never broken adjacent to the arrow in a lambda, except that a break may come immediately after the arrow if the body of the lambda consists of a single unbraced expression

MyLambda<String, Long, Object> lambda =

(String label, Long value, Object obj) -> {

...

};

Predicate<String> predicate = str ->

longExpressionInvolving(str);

* Line wrapping for if statements should generally use the 8-space rule, since conventional (4 space) indentation makes seeing the body difficult. 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;

## **Comments**

Java programs can have two kinds of comments:

* implementation comments /\*...\*/, and //. These are meant for commenting out code.

/\*

\* This is // And so /\* Or you can

\* okay. // is this. \* even do this. \*/

\*/

if (a == 2) {

return TRUE; /\* special case \*/

} else {

return isPrime(a); /\* works only for odd a \*/

}

if (foo > 1) {

// Do a double-flip.

...

}

else {

return false; // Explain why here.

}

* documentation comments. /\*\*...\*/. These are meant to describe the specification of the code

/\*\*

\* The Example class provides ...

\*/

public class Example { ...

/\*\*

\* Returns an Image object that can then be painted on the screen.

\* The url argument must specify an absolute [{@link URL}](https://www.oracle.com/technetwork/java/javase/documentation/index-137868.html#{@link}). The name

\* argument is a specifier that is relative to the url argument.

\* <p>

\* This method always returns immediately, whether or not the

\* image exists. When this applet attempts to draw the image on

\* the screen, the data will be loaded. The graphics primitives

\* that draw the image will incrementally paint on the screen.

\*

\* [@param](https://www.oracle.com/technetwork/java/javase/documentation/index-137868.html#@param) url an absolute URL giving the base location of the image

\* [@param](https://www.oracle.com/technetwork/java/javase/documentation/index-137868.html#@param) name the location of the image, relative to the url argument

\* [@return](https://www.oracle.com/technetwork/java/javase/documentation/index-137868.html#@return) the image at the specified URL

\* [@see](https://www.oracle.com/technetwork/java/javase/documentation/index-137868.html#@see) Image

\*/

public Image getImage(URL url, String name) {

try {

return getImage(new URL(url, name));

} catch (MalformedURLException e) {

return null;

}

}

## **Formatting**

* No line break before the opening brace.
* Line break after the opening brace.
* Line break before the closing brace.
* Line break after the closing brace, only if that brace terminates a statement or terminates the body of a method, constructor, or named class. For example, there is no line break after the brace if it is followed by else or a comma.

return () -> {

while (condition()) {

method();

}

};

return new MyClass() {

@Override public void method() {

if (condition()) {

try {

something();

} catch (ProblemException e) {

recover();

}

} else if (otherCondition()) {

somethingElse();

} else {

lastThing();

}

}

};

#### Empty blocks: may be concise

An empty block closed immediately after it is opened, with no characters or line break in between ({}), **unless** it is part of a *multi-block statement* (one that directly contains multiple blocks: if/else or try/catch/finally).

// This is acceptable

void doNothing() {}

// This is equally acceptable

void doNothingElse() {

}

// This is not acceptable: No concise empty blocks in a multi-block statement

try {

doSomething();

} catch (Exception e) {}

## **Declarations**

* 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

int foo, fooarray[]; //WRONG!

* Try to initialize local variables where they're declared. The only reason not to initialize a variable where it's declared is if the initial value depends on some computation occurring first.
* Put declarations only at the beginning of blocks. Don't wait to declare variables until their first use

void myMethod() {

int int1 = 0; // beginning of method block

if (condition) {

int int2 = 0; // beginning of "if" block

...

}

}

* Avoid local declarations that hide declarations at higher levels

int count;

...

myMethod() {

if (condition) {

int count = 0; // AVOID!

...

}

...

}

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

* No space 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 "{"
* Methods are separated by a blank line

class Sample extends Object {

int ivar1;

int ivar2;

Sample(int i, int j) {

ivar1 = i;

ivar2 = j;

}

int emptyMethod() {}

...

}

## **Statements**

Each line should contain at most one statement. Example:

argv++; // Correct

argc--; // Correct

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

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);

### 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*;

### 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);

A while statement should have the following form:

while (condition) {

statements;

}

An empty while statement should have the following form:

while (condition);

### do-while Statements

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

do {

statements;

} while (condition);

### switch Statements

Every time a case falls through (doesn't 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 later another case is added.

**Exception:** A switch statement for an enum type may omit the default statement group, if it includes explicit cases covering all possible values of that type

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;

}

### 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;

}

### Caught exceptions: not ignored

Except as noted below, it is very rarely correct to do nothing in response to a caught exception. (Typical responses are to log it, or if it is considered "impossible", rethrow it as an AssertionError.)

When it truly is appropriate to take no action whatsoever in a catch block, the reason this is justified is explained in a comment.

try {

int i = Integer.parseInt(response);

return handleNumericResponse(i);

} catch (NumberFormatException ok) {

// it's not numeric; that's fine, just continue

}

return handleTextResponse(response);

## **Naming** **Conventions**

* 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.

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.

Package names are all lowercase, with consecutive words simply concatenated together (no underscores)

com.example.deepspace

com.sun.eng

com.apple.quicktime.v2

edu.cmu.cs.bovik.cheese

com.example.deepSpace WRONG!

com.example.deep\_space. WRONG!

#### Class names

Class names are written in [**UpperCamelCase**](https://google.github.io/styleguide/javaguide.html#s5.3-camel-case).

Class names are typically nouns or noun phrases. For example, Character or **ImmutableList**. Interface names may also be nouns or noun phrases (for example, **List**), but may sometimes be adjectives or adjective phrases instead (for example, **Readable**).There are no specific rules or even well-established conventions for naming annotation types.

Test classes are named starting with the name of the class they are testing, and ending with Test. For example, HashTest or HashIntegrationTest

#### Method names

Method names are written in [**lowerCamelCase**](https://google.github.io/styleguide/javaguide.html#s5.3-camel-case).

Method names are typically verbs or verb phrases.

For example, sendMessage(),Stop(),run(),runFast(),getBackground();

Underscores may appear in JUnit *test* method names to separate logical components of the name, with *each* component written in [lowerCamelCase](https://google.github.io/styleguide/javaguide.html#s5.3-camel-case). One typical pattern is <methodUnderTest>\_<state>, for example pop\_emptyStack. There is no One Correct Way to name test methods.

#### Constant names

Constant names use CONSTANT\_CASE: all uppercase letters, with each word separated from the next by a single underscore. But what is a constant, exactly?

Constants are static final fields whose contents are deeply immutable and whose methods have no detectable side effects. This includes primitives, Strings, immutable types, and immutable collections of immutable types

**// Constants**

static final int NUMBER = 5;

static final ImmutableList<String> NAMES = ImmutableList.of("Ed", "Ann");

static final ImmutableMap<String, Integer> AGES = ImmutableMap.of("Ed", 35, "Ann", 32);

static final Joiner COMMA\_JOINER = Joiner.on(','); // because Joiner is immutable

static final SomeMutableType[] EMPTY\_ARRAY = {};

enum SomeEnum { ENUM\_CONSTANT }

**// Not constants**

Non-constant field names (static or otherwise) are written in [lowerCamelCase](https://google.github.io/styleguide/javaguide.html#s5.3-camel-case).

hese names are typically nouns or noun phrases. For example, computedValues or index

static String nonFinal = "non-final";

final String nonStatic = "non-static";

static final Set<String> mutableCollection = new HashSet<String>();

static final ImmutableSet<SomeMutableType> mutableElements = ImmutableSet.of(mutable);

static final ImmutableMap<String, SomeMutableType> mutableValues =

ImmutableMap.of("Ed", mutableInstance, "Ann", mutableInstance2);

static final Logger logger = Logger.getLogger(MyClass.getName());

static final String[] nonEmptyArray = {"these", "can", "change"};

**Parameter names** are written in **lowerCamelCase**.

One-character parameter names in public methods should be avoided.

**Local variable** names are written in **lowerCamelCase**.

Even when final and immutable, local variables are not considered to be constants, and should not be styled as constants.

|  |  |  |
| --- | --- | --- |
| **Prose form** | **Correct** | **Incorrect** |
| "XML HTTP request" | XmlHttpRequest | XMLHTTPRequest |
| "new customer ID" | newCustomerId | newCustomerID |
| "inner stopwatch" | innerStopwatch | innerStopWatch |
| "supports IPv6 on iOS?" | supportsIpv6OnIos | supportsIPv6OnIOS |
| "YouTube importer" | YouTubeImporter |  |
| YoutubeImporter\* |

#### **Modifiers**

Class and member modifiers, when present, appear in the order recommended by the Java Language Specification:

public protected private abstract default static final transient volatile synchronized native strictfp

#### **Numeric Literals**

long-valued integer literals use an uppercase L suffix, never lowercase (to avoid confusion with the digit 1). For example, 3000000000L rather than 3000000000l.

#### **Annotations**

Annotations applying to a class, method or constructor appear immediately after the documentation block and each annotation is listed on a line of its own (that is, one annotation per line)

@Override

@Nullable

public String getNameIfPresent() { ... }

**Exception:** A *single* parameterless annotation *may* instead appear together with the first line of the signature, for example:

@Override public int hashCode() { ... }

Annotations applying to a field also appear immediately after the documentation block, but in this case, *multiple* annotations (possibly parameterized) may be listed on the same line

@Partial @Mock DataLoader loader;

#### **@Override: always used**

A method is marked with the @Override annotation whenever it is legal. This includes a class method overriding a superclass method, a class method implementing an interface method, and an interface method respecifying a superinterface method.

**Exception:** @Override may be omitted when the parent method is @Deprecated.

### Static members: qualified using class

When a reference to a static class member must be qualified, it is qualified with that class's name, not with a reference or expression of that class's type.

Foo aFoo = ...;

Foo.aStaticMethod(); // good

aFoo.aStaticMethod(); // bad

somethingThatYieldsAFoo().aStaticMethod(); // very bad

**Finalizers: not used**

It is **extremely rare** to override Object.finalize.

#### **Enum classes**

After each comma that follows an enum constant, a line break is optional. Additional blank lines (usually just one) are also allowed. This is one possibility:

private enum Answer {

YES {

@Override public String toString() {

return "yes";

}

},

NO,

MAYBE

}

An enum class with no methods and no documentation on its constants may optionally be formatted as if it were an array initializer

private enum Suit { CLUBS, HEARTS, SPADES, DIAMONDS }

**Coding Standards:**

* Swagger Annotations should have proper information about API

@ApiOperation(value = "…….", notes = "……..")

@ApiResponses({ //swagger - describe return status code

@ApiResponse(code=200,message="Success. Request completed."),

@ApiResponse(code=400,message="BAD …"),

@ApiResponse(code=404,message="Not Found.."),

@ApiResponse(code=500,message="Internal Server error."),

* Input parameters for Controller methods should be validated

// Validate required fields

List<Errors> errors = validate(input);

**if** (errors!= **null** && ! errors.isEmpty()) { methodInfo.append("HttpStatus:").append(HttpStatus.***BAD\_REQUEST***.name());

* All mandatory parameters of public methods should be validated for null check

**if** (Util.*isNullOrEmpty*(list)) {

**throw** **new** IllegalArgumentException("list should not be null/empty");

}

* Log method names and method info in all method loggers

**final** **long** startTimeInMS = System.*currentTimeMillis*();

StringBuilder methodInfo = **new** StringBuilder();

methodInfo.append(" targetCompanyId:").append(targetCompanyId);

LOGGER.info(methodName + methodInfo.toString() + " Started getting top 5 correlation data ");

* Log processing time of each method at the end of method

SystemUtil.timeSpent(methodName, methodInfo.toString(), startTimeInMS, Level.INFO);

**Coding Standards:**

Code Style “Best Practices”:

1. Omit the Java keyword `this` when it is optional
2. Use enums over collections of strings for enumerated value sets.
   1. These should be a ‘single source of truth’ as to what values can be in a field.
3. Use `log` to define logging object at the top of each class (slf4j) – suggest using Lombok annotation
4. Use descriptive names over abbreviations
5. When possible, avoid excessive comments by making code “self-documenting”
6. When possible, use shorter methods and more of them – methods should “do one thing”
7. Minimize use of blank lines – they should occur rarely inside a function.
8. Handle exit conditions quickly
9. Avoid negative logic when possible
10. Avoid nested conditionals when possible
11. Complex conditionals should be simplified and moved to their own method if they aren’t easily readable. Data objects are a great place for this if they are universal conditions
12. Log appropriately – use correct levels, templated style, avoid logging PII
13. Suggest using Lombok when appropriate

**Advantages**

Google has a great wealth of expertise in software development. By adopting their standards, we absolve ourselves of the maintenance overhead of debating changes as they occur while relying on a trusted expert.

By using Google’s style as a standard, we have an authoritative source of “right vs wrong”.

Automatic formatters available for Eclipse and IntelliJ, reducing manual formatting overhead. VSCode uses Eclipse code formatter under the hood for Java, so our three major IDEs are covered.

**Disadvantages**

Many of our existing files have sharp differences from the standard. This will require an adoption path to avoid every merge being confusing.

Since the style is in many ways prescriptive, it will require adjustment for many of us.

**Suggested Adoption Path**

Import the style to your IDE, turn on auto formatting on save (see below).

Each scrum team should coordinate a single party to do this: for each project, prepare a single merge request with all files in a project, auto format, and save (you can use the menus to auto format or just add a space somewhere in the file and then save – the `format on save` will pick this up). These changes should be in their own merge request containing nothing but format changes. This will isolate the formatting changes rather than intermingling them with changes in functionality.

**Maintaining Code Style in Repos**

If anything besides our editors configured with auto formatting are used to edit a file, they should be opened and formatted with an appropriate IDE before committing.

If improperly formatted merge requests are submitted after this conversion, they should not be approved.

**Style details**

Guide: <http://google.github.io/styleguide/>

Style guide repo: <https://github.com/google/styleguide>

**Instructions to enable automatic formatting on save**

From git bash or other command line: “git clone https://github.com/google/styleguide.git”

Eclipse:

* + - * 1. Window -> Preferences
        2. Java -> Code Style -> Formatter,
        3. Import, select eclipse-java-google-style inside cloned repo
        4. Java -> Editor -> Save Actions
        5. Check “Perform the selected actions on save, Format source Code, Format all lines, Organize Imports