Java Style Guide

Adapted from Google Java Style Guide - https://google.github.io/styleguide/javaguide.html

Table Of Contents

Non-ASCII characters

When non-ASCII characters are required, either the actual Unicode character (e.g. ∞) or the equivalent Unicode escape (e.g. \u221e) is used. The choice depends only on which makes the code easier to read and understand. Where useful, an explanatory comment should be used.

Source file structure

A source file consists of, in order:

1. Package statement
2. Import statements
3. Exactly one top-level class

Exactly one blank line separates each section that is present.

No wildcard imports

Wildcard imports, static or otherwise, are not used.

Class declaration

Each top-level class resides in a source file of its own. When a class has multiple constructors, or multiple methods with the same name, these appear sequentially, with no intervening members (not even private ones).

Formatting

Terminology Note: block-like construct refers to the body of a class, method or constructor.

Braces

Braces are used where optional

Braces are used with if, else, for, do and while statements, even when the body is empty or contains only a single statement.

Nonempty blocks

- 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.

```
@Override public void method() {
  if (condition()) {
    someThing();
  } else {
    otherThing();
  }
}
```

Empty blocks: may be concise

An empty block or block-like construct may be closed immediately after it is opened, with no characters or line break in between ({}). E.g. void doNothing() {}

Block indentation

Each time a new block or block-like construct is opened, the indent increases by four spaces or a tab. When the block ends, the indent returns to the previous indent level. The indent level applies to both code and comments throughout the block.

Line break indentation

When line-wrapping, each line after the first (each continuation line) is indented +1 from the original line.

One statement per line

Each statement is followed by a line break.

Whitespace

Vertical Whitespace

A single blank line appears:

1. Between consecutive members (or initializers) of a class: constructors, methods, nested classes, static initializers, instance initializers.
2. Between statements and fields, as needed to organize the code into logical subsections.
3. Optionally before the first member or after the last member of the class.

Horizontal whitespace

Beyond where required by the language or other style rules, a single ASCII space also appears in the following places:

1. Separating any reserved word, such as `if`, `for` or `catch`, from an open parenthesis (`(`) that follows it on that line
2. Separating any reserved word, such as `else` or `catch`, from a closing curly brace (`}`) that precedes it on that line
3. Before any open curly brace (`{`), with two exceptions:
    - `@SomeAnnotation({a, b})` (no space is used)
    - `String[][] x = {{"foo"}};` (no space is required between `{{`, by item 8 below)
4. On both sides of any binary or ternary operator. This also applies to the following "operator-like" symbols:
    - the ampersand in a conjunctive type bound: `<T extends Foo & Bar>`
    - the pipe for a catch block that handles multiple exceptions: `catch (FooException | BarException e)`
    - the colon (`:`) in an enhanced `for` ("foreach") statement
    - the arrow in a lambda expression: `(String str) -> str.length()`
5. but not
    - the two colons (`::`) of a method reference, which is written like `Object::toString`
    - the dot separator (`.`), which is written like `object.toString()`
6. On both sides of the double slash (`//`) that begins an end-of-line comment. Here, two spaces should be used before the double slash.
7. Between the type and variable of a declaration: `List<String> list`

Grouping parentheses

Optional grouping parentheses should be used where it will reduce the change the code will be misinterpreted.

Specific constructs

Enum classes

After each comma that follows an enum constant, a line break should be used.

```
private enum Answer {
  YES {
    @Override public String toString() {
      return "yes";
    }
  },
  NO,
  MAYBE
}
```

Variable declarations

Multiple variables per declaration

Variable declarations may declare more than one variable where appropriate. E.g. int a, b;

Declared when needed

Local variables are declared close to the point they are first used (within reason), to minimize their scope. Local variable declarations typically have initializers, or are initialized immediately after declaration.

Arrays

Array initializers

The following styles should be used unless another is found to be much more appropriate:

```
new int[] first {
  0, 1, 2, 3
}

new int[] second {
  0,
  1,
  2,
  3
}
```

No C-style array declarations

The square brackets form a part of the *type*, not the variable: `String[] args`, not `String args[]`.

Switch statements

Terminology Note: Inside the braces of a switch block are one or more statement groups. Each statement group consists of one or more switch labels (either `case FOO:` or `default:`), followed by one or more statements.

Indentation

As with any other block, the contents of a switch block are indented +4 or tab. After a switch label, a new line appears, and the indentation level is increased +4 or tab, exactly as if a block were being opened. The following switch label returns to the previous indentation level, as if a block had been closed.

Fall-through: commented

Within a switch block, each statement group either terminates abruptly (with a `break`, `continue`, `return` or thrown exception), or is marked with a comment to indicate that execution will or *might* continue into the next statement group. Any comment that communicates the idea of fall-through is sufficient (typically `// fall through`). This special comment is not required in the last statement group of the switch block. Example:

```
switch (input) {
  case 1:
  case 2:
    prepareOneOrTwo();
    // fall through
  case 3:
    handleOneTwoOrThree();
    break;
  default:
    handleLargeNumber(input);
}
```

Notice that no comment is needed after `case 1:`, only at the end of the statement group.

The `default` case is present

Each switch statement includes a `default` statement group, even if it contains no code.

Annotations

Annotations appear immediately after any documentation block, and each annotation is listed on a line of its own (that is, one annotation per line). The indentation level is not increased. Example:

```
@Override
@Nullable
public String getNameIfPresent() { ... }
```

Comments

Block comment style

Block comments are indented at the same level as the surrounding code. They may be in `/* ... */` style or `// ...` style. For multi-line `/* ... */` comments, subsequent lines must start with `*` aligned with the `*` on the previous line.

```
/*
 * This is          // And so
 * okay.            // is this.
 */
```

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`.

Naming

Package names

Package names are all lowercase, with consecutive words simply concatenated together (no underscores). For example, `com.example.deepspace`, not `com.example.deepSpace` or `com.example.deep_space`.

Class names

Class names are written in UpperCamelCase. 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`). ***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.
Method names are typically verbs or verb phrases. For example, sendMessage or stop.
Underscores appear in JUnit test method names to separate the method name, simple description, and expected outcome.
`test<MethodUnderTest>_<descripiton>_<expectedOutcome>`, for example: `testSaveData_giveInvalidData_exceptionThrown`.

Constant names

Constant names use CONSTANT_CASE: all uppercase letters, with words separated by underscores.

Non-constant field, local variable, and parameter names

These are written in lowerCamelCase. These names are typically nouns or noun phrases. For example, `computedValues` or `index`.

Type variable names

Each type variable is named in one of two styles:

- A single capital letter, optionally followed by a single numeral (such as E, T, X, T2)
- A name in the form used for classes followed by the capital letter T (examples: RequestT, FooBarT).

Programming Practices

@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.

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);
```

Exception: In tests, a caught exception may be ignored without comment if its name is or begins with expected. The following is a very common idiom for ensuring that the code under test does throw an exception of the expected type, so a comment is unnecessary here.

```
try {
  emptyStack.pop();
  fail();
} catch (NoSuchElementException expected) {
}
```

Javadoc

Formatting

General form

The basic formatting of Javadoc blocks is as seen in this example:

```
/**
 * Multiple lines of Javadoc text are written here,
 * wrapped normally...
 */
public int method(String p1) { ... }
```

... or in this single-line example:

```
/** An especially short bit of Javadoc. */
```

Paragraphs

One blank line—that is, a line containing only the aligned leading asterisk (*)—appears between paragraphs, and before the group of "at-clauses" if present.

At-clauses

Any of the standard "at-clauses" that are used appear in the order `@param`, `@return`, `@throws`, `@deprecated`, and these four types never appear with an empty description. When an at-clause doesn't fit on a single line, continuation lines are indented four spaces or a tab from the position of the `@`.

The summary fragment

The Javadoc for each class and member begins with a brief summary fragment. This fragment is very important: it is the only part of the text that appears in certain contexts such as class and method indexes.
This is a fragment—a noun phrase or verb phrase, not a complete sentence. It does **not** begin with `A {@code Foo} is a...`, or `This method returns...`. E.g. `/** Returns the customer ID. */`.

Where Javadoc is used

At the minimum, Javadoc is present for every public class, and every public or protected member of such a class, with a few exceptions noted below. Other classes and members have Javadoc as needed or desired. Whenever an implementation comment would be used to define the overall purpose or behavior of a class or member, that comment is written as Javadoc instead (using `/**`).

Exceptions

Javadoc is optional for "simple, obvious" methods like getFoo, in cases where there really and truly is nothing else worthwhile to say but "Returns the foo". However Javadoc should be included if it is not clear exactly what “foo” is.
Javadoc is not always present on a method that overrides a supertype method.