Google Java Style Guide

Table of Contents

1 Introduction

This document serves as the **complete** definition of Google's coding standards for source code in the Java™ Programming Language. A Java source file is described as being *in Google Style* if and only if it adheres to the rules herein.

Like other programming style guides, the issues covered span not only aesthetic issues of formatting, but other types of conventions or coding standards as well. However, this document focuses primarily on the **hard-and-fast rules** that we follow universally, and avoids giving *advice* that isn't clearly enforceable (whether by human or tool).

1.1 Terminology notes

In this document, unless otherwise clarified:

1. The term *class* is used inclusively to mean an "ordinary" class, record class, enum class, interface or annotation type (`@interface`).
2. The term *member* (of a class) is used inclusively to mean a nested class, field, method, *or constructor*; that is, all top-level contents of a class except initializers and comments.
3. The term *comment* always refers to *implementation* comments. We do not use the phrase "documentation comments", and instead use the common term "Javadoc."

Other "terminology notes" will appear occasionally throughout the document.

1.2 Guide notes

Example code in this document is **non-normative**. That is, while the examples are in Google Style, they may not illustrate the *only* stylish way to represent the code. Optional formatting choices made in examples should not be enforced as rules.

2 Source file basics

2.1 File name

For a source file containing classes, the file name consists of the case-sensitive name of the top-level class (of which there is exactly one), plus the `.java` extension.

2.2 File encoding: UTF-8

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

2.3 Special characters

2.3.1 Whitespace characters

Aside from the line terminator sequence, the **ASCII horizontal space character** (**0x20**) is the only whitespace character that appears anywhere in a source file. This implies that:

1. All other whitespace characters in string and character literals are escaped.
2. Tab characters are **not** used for indentation.

2.3.2 Special escape sequences

For any character that has a special escape sequence (`\b` , `\t` , `\n` , `\f` , `\r` , `\s` , `\"` , `\'` and `\\`), that sequence is used rather than the corresponding octal (e.g. `\012`) or Unicode (e.g. `\u000a`) escape.

2.3.3 Non-ASCII characters

For the remaining non-ASCII characters, 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**, although Unicode escapes outside string literals and comments are strongly discouraged.

> **Tip:** In the Unicode escape case, and occasionally even when actual Unicode characters are used, an explanatory comment can be very helpful.

Examples:

Example	Discussion
`String unitAbbrev = "μs";`	Best: perfectly clear even without a comment.
`String unitAbbrev = "\u03bcs"; // "μs"`	Allowed, but there's no reason to do this.
`String unitAbbrev = "\u03bcs"; // Greek letter mu, "s"`	Allowed, but awkward and prone to mistakes.
`String unitAbbrev = "\u03bcs";`	Poor: the reader has no idea what this is.
`return '\ufeff' + content; // byte order mark`	Good: use escapes for non-printable characters, and comment if necessary.

> **Tip:** Never make your code less readable simply out of fear that some programs might not handle non-ASCII characters properly. If that should happen, those programs are **broken** and they must be **fixed**.

3 Source file structure

An ordinary source file consists of, **in order**:

1. License or copyright information, if present
2. Package statement
3. Import statements
4. Exactly one top-level class

Exactly one blank line separates each section that is present.

A `package-info.java` file is the same, but without the top-level class.

A `module-info.java` file does not contain a package statement and replaces the single top-level class with a module declaration, but otherwise follows the same structure.

3.1 License or copyright information, if present

If license or copyright information belongs in a file, it belongs here.

3.2 Package statement

The package statement is **not line-wrapped**. The column limit (Section 4.4, Column limit: 100) does not apply to package statements.

3.3 Import statements

3.3.1 No wildcard imports

Wildcard imports, static or otherwise, **are not used**.

3.3.2 No line-wrapping

Import statements are **not line-wrapped**. The column limit (Section 4.4, Column limit: 100) does not apply to import statements.

3.3.3 Ordering and spacing

Imports are ordered as follows:

1. All static imports in a single block.
2. All non-static imports in a single block.

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.

Within each block the imported names appear in ASCII sort order. (**Note:** this is not the same as the import *statements* being in ASCII sort order, since '.' sorts before ';'.)

3.3.4 No static import for classes

Static import is not used for static nested classes. They are imported with normal imports.

3.4 Class declaration

3.4.1 Exactly one top-level class declaration

Each top-level class resides in a source file of its own.

3.4.2 Ordering of class contents

The order you choose for the members and initializers of your class can have a great effect on learnability. However, there's no single correct recipe for how to do it; different classes may order their contents in different ways.

What is important is that each class uses ***some* logical order**, which its maintainer could explain if asked. For example, new methods are not just habitually added to the end of the class, as that would yield "chronological by date added" ordering, which is not a logical ordering.

3.4.2.1 Overloads: never split

Methods of a class that share the same name appear in a single contiguous group with no other members in between. The same applies to multiple constructors (which always have the same name). This rule applies even when modifiers such as `static` or `private` differ between the methods.

3.5 Module declaration

3.5.1 Ordering and spacing of module directives

Module directives are ordered as follows:

1. All `requires` directives in a single block.
2. All `exports` directives in a single block.
3. All `opens` directives in a single block.
4. All `uses` directives in a single block.
5. All `provides` directives in a single block.

A single blank line separates each block that is present.

4 Formatting

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

Note that, by Section 4.8.3.1 on array initializers, any array initializer *may* optionally be treated as if it were a block-like construct.

4.1 Braces

4.1.1 Use of optional braces

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

Other optional braces, such as those in a lambda expression, remain optional.

4.1.2 Nonempty blocks: K & R style

Braces follow the Kernighan and Ritchie style ("Egyptian brackets") for *nonempty* blocks and block-like constructs:

- No line break before the opening brace, except as detailed below.
- 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.

Exception: In places where these rules allow a single statement ending with a semicolon (`;`), a block of statements can appear, and the opening brace of this block is preceded by a line break. Blocks like these are typically introduced to limit the scope of local variables.

Examples:

```
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();
```

```
      doSomething();
    }
    {
      int x = foo();
      frob(x);
    }
  }
};
```

A few exceptions for enum classes are given in Section 4.8.1, Enum classes.

4.1.3 Empty blocks: may be concise

An empty block or block-like construct may be in K & R style (as described in Section 4.1.2). Alternatively, it may be 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`).

Examples:

```
// This is acceptable
void doNothing() {}

// This is equally acceptable
void doNothingElse() {
}
```

```
// This is not acceptable: No concise empty blocks in a multi-block stateme
try {
  doSomething();
} catch (Exception e) {}
```

4.2 Block indentation: +2 spaces

Each time a new block or block-like construct is opened, the indent increases by two spaces. When the block ends, the indent returns to the previous indent level. The indent level applies to both code and comments throughout the block. (See the example in Section 4.1.2, Nonempty blocks: K & R Style.)

4.3 One statement per line

Each statement is followed by a line break.

4.4 Column limit: 100

Java code has a column limit of 100 characters. A "character" means any Unicode code point. Except as noted below, any line that would exceed this limit must be line-wrapped, as explained in Section 4.5, Line-wrapping.

> Each Unicode code point counts as one character, even if its display width is greater or less. For example, if using fullwidth characters, you may choose to wrap the line earlier than where this rule strictly requires.

Exceptions:

1. Lines where obeying the column limit is not possible (for example, a long URL in Javadoc, or a long JSNI method reference).
2. `package` and `import` statements (see Sections 3.2 Package statement and 3.3 Import statements).
3. Contents of text blocks.
4. Command lines in a comment that may be copied-and-pasted into a shell.
5. Very long identifiers, on the rare occasions they are called for, are allowed to exceed the column limit. In that case, the valid wrapping for the surrounding code is as produced by google-java-format.

4.5 Line-wrapping

Terminology Note: When code that might otherwise legally occupy a single line is divided into multiple lines, this activity is called *line-wrapping*.

There is no comprehensive, deterministic formula showing *exactly* how to line-wrap in every situation. Very often there are several valid ways to line-wrap the same piece of code.

> **Note:** While the typical reason for line-wrapping is to avoid overflowing the column limit, even code that would in fact fit within the column limit *may* be line-wrapped at the author's discretion.

> **Tip:** Extracting a method or local variable may solve the problem without the need to line-wrap.

4.5.1 Where to break

The prime directive of line-wrapping is: prefer to break at a **higher syntactic level**. Also:

1. When a line is broken at a *non-assignment* operator the break comes *before* the symbol. (Note that this is not the same practice used in Google style for other languages, such as C++ and JavaScript.)
   - This also applies to the following "operator-like" symbols:
     - the dot separator (`.`)
     - the two colons of a method reference (`::`)
     - an ampersand in a type bound (`<T extends Foo & Bar>`)
     - a pipe in a catch block (`catch (FooException | BarException e)`).
2. When a line is broken at an *assignment* operator the break typically comes *after* the symbol, but either way is acceptable.

- This also applies to the "assignment-operator-like" colon in an enhanced `for` ("foreach") statement.

3. A method, constructor, or record-class name stays attached to the open parenthesis (`(`) that follows it.
4. A comma (`,`) stays attached to the token that precedes it.
5. A line is never broken adjacent to the arrow in a lambda or a switch rule, except that a break may come immediately after the arrow if the text following it consists of a single unbraced expression. Examples:

```
MyLambda<String, Long, Object> lambda =
    (String label, Long value, Object obj) -> {
      ...
    };

Predicate<String> predicate = str ->
    longExpressionInvolving(str);

switch (x) {
  case ColorPoint(Color color, Point(int x, int y)) ->
      handleColorPoint(color, x, y);
  ...
}
```

> **Note:** The primary goal for line wrapping is to have clear code, *not necessarily* code that fits in the smallest number of lines.

4.5.2 Indent continuation lines at least +4 spaces

When line-wrapping, each line after the first (each *continuation line*) is indented at least +4 from the original line.

When there are multiple continuation lines, indentation may be varied beyond +4 as desired. In general, two continuation lines use the same indentation level if and only if they begin with syntactically parallel elements.

Section 4.6.3 on Horizontal alignment addresses the discouraged practice of using a variable number of spaces to align certain tokens with previous lines.

4.6 Whitespace

4.6.1 Vertical Whitespace

A single blank line always appears:

1. *Between* consecutive members or initializers of a class: fields, constructors, methods, nested classes, static initializers, and instance initializers.
   - **Exception:** A blank line between two consecutive fields (having no other code

between them) is optional. Such blank lines are used as needed to create *logical groupings* of fields.

- **Exception:** Blank lines between enum constants are covered in Section 4.8.1.

2. As required by other sections of this document (such as Section 3, Source file structure, and Section 3.3, Import statements).

A single blank line may also appear anywhere it improves readability, for example between statements to organize the code into logical subsections. A blank line before the first member or initializer, or after the last member or initializer of the class, is neither encouraged nor discouraged.

Multiple consecutive blank lines are permitted, but never required (or encouraged).

4.6.2 Horizontal whitespace

Beyond where required by the language or other style rules, and apart from literals, comments and Javadoc, a single ASCII space also appears in the following places **only**.

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 9 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()` or switch rule: `case "FOO" -> bar();`

   but not
   - the two colons (`::`) of a method reference, which is written like `Object::toString`
   - the dot separator (`.`), which is written like `object.toString()`
5. After `,:;` or the closing parenthesis (`)`) of a cast
6. Between any content and a double slash (`//`) which begins a comment. Multiple spaces are allowed.
7. Between a double slash (`//`) which begins a comment and the comment's text. Multiple spaces are allowed.
8. Between the type and variable of a declaration: `List<String> list`
9. *Optional* just inside both braces of an array initializer
   - `new int[] {5, 6}` and `new int[] { 5, 6 }` are both valid
10. Between a type annotation and `[]` or `...`.

This rule is never interpreted as requiring or forbidding additional space at the start or end of a line; it addresses only *interior* space.

4.6.3 Horizontal alignment: never required

Terminology Note: *Horizontal alignment* is the practice of adding a variable number of additional spaces in your code with the goal of making certain tokens appear directly below certain other tokens on previous lines.

This practice is permitted, but is **never required** by Google Style. It is not even required to *maintain* horizontal alignment in places where it was already used.

Here is an example without alignment, then using alignment:

```
private int x; // this is fine
private Color color; // this too

private int   x;      // permitted, but future edits
private Color color;  // may leave it unaligned
```

> **Tip:** Alignment can aid readability, but attempts to preserve alignment for its own sake create future problems. For example, consider a change that touches only one line. If that change disrupts the previous alignment, it's important **not** to introduce additional changes on nearby lines simply to realign them. Introducing formatting changes on otherwise unaffected lines corrupts version history, slows down reviewers, and exacerbates merge conflicts. These practical concerns take priority over alignment.

4.7 Grouping parentheses: recommended

Optional grouping parentheses are omitted only when author and reviewer agree that there is no reasonable chance the code will be misinterpreted without them, nor would they have made the code easier to read. It is *not* reasonable to assume that every reader has the entire Java operator precedence table memorized.

4.8 Specific constructs

4.8.1 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 (see Section 4.8.3.1 on array initializers).

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

Since enum classes *are classes*, all other rules for formatting classes apply.

4.8.2 Variable declarations

4.8.2.1 One variable per declaration

Every variable declaration (field or local) declares only one variable: declarations such as `int a, b;` are not used.

Exception: Multiple variable declarations are acceptable in the header of a `for` loop.

4.8.2.2 Declared when needed

Local variables are **not** habitually declared at the start of their containing block or block-like construct. Instead, 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.

4.8.3 Arrays

4.8.3.1 Array initializers: can be "block-like"

Any array initializer may *optionally* be formatted as if it were a "block-like construct." For example, the following are all valid (**not** an exhaustive list):

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

4.8.3.2 No C-style array declarations

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

4.8.4 Switch statements and expressions

For historical reasons, the Java language has two distinct syntaxes for `switch`, which we can call *old-style* and *new-style*. New-style switches use an arrow (`->`) after the switch labels, while old-style switches use a colon (`:`).

Terminology Note: Inside the braces of a *switch block* are either one or more *switch rules* (new-style); or one or more *statement groups* (old-style). A *switch rule* consists of a *switch label* (`case ...` or `default`) followed by `->` and an expression, block, or `throw`. A statement group consists of one or more switch labels each followed by a colon, then one or more statements, or, for the *last* statement group, *zero* or more statements. (These definitions match the Java Language Specification, §14.11.)

4.8.4.1 Indentation

As with any other block, the contents of a switch block are indented +2. Each switch label starts with this +2 indentation.

In a new-style switch, a switch rule can be written on a single line if it otherwise follows Google style. (It must not exceed the column limit, and if it contains a non-empty block then there must be a line break after `{`.) The line-wrapping rules of Section 4.5 apply, including the +4 indent for continuation lines. For a switch rule with a non-empty block after the arrow, the same rules apply as for blocks elsewhere: lines between `{` and `}` are indented a further +2 relative to the line with the switch label.

```
switch (number) {
  case 0, 1 -> handleZeroOrOne();
  case 2 ->
      handleTwoWithAnExtremelyLongMethodCallThatWouldNotFitOnTheSameLine();
  default -> {
    logger.atInfo().log("Surprising number %s", number);
    handleSurprisingNumber(number);
  }
}
```

In an old-style switch, the colon of each switch label is followed by a line break. The statements within a statement group start with a further +2 indentation.

4.8.4.2 Fall-through: commented

Within an old-style 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.

There is no fall-through in new-style switches.

4.8.4.3 Exhaustiveness and presence of the `default` label

The Java language requires switch expressions and many kinds of switch statements to be *exhaustive*. That effectively means that every possible value that could be switched on will be matched by one of the switch labels. A switch is exhaustive if it has a `default` label, but also for example if the value being switched on is an enum and every value of the enum is matched by a switch label. Google Style requires *every* switch to be exhaustive, even those where the language itself does not require it. This may require adding a `default` label, even if it contains no code.

4.8.4.4 Switch expressions

Switch expressions must be new-style switches:

```
  return switch (list.size()) {
    case 0 -> "";
    case 1 -> list.getFirst();
    default -> String.join(", ", list);
  };
```

4.8.5 Annotations

4.8.5.1 Type-use annotations

Type-use annotations appear immediately before the annotated type. An annotation is a type-use annotation if it is meta-annotated with `@Target(ElementType.TYPE_USE)`. Example:

```
final @Nullable String name;

public @Nullable Person getPersonByName(String name);
```

4.8.5.2 Class, package, and module annotations

Annotations applying to a class, package, or module declaration appear immediately after the documentation block, and each annotation is listed on a line of its own (that is, one annotation per line). These line breaks do not constitute line-wrapping (Section 4.5, Line-wrapping), so the indentation level is not increased. Examples:

```
/** This is a class. */
@Deprecated
@CheckReturnValue
public final class Frozzler { ... }
```

```
/** This is a package. */
@Deprecated
@CheckReturnValue
package com.example.frozzler;
```

```
/** This is a module. */
@Deprecated
@SuppressWarnings("CheckReturnValue")
module com.example.frozzler { ... }
```

4.8.5.3 Method and constructor annotations

The rules for annotations on method and constructor declarations are the same as the previous section. Example:

```
@Deprecated
@Override
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() { ... }
```

4.8.5.4 Field annotations

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; for example:

```
@Partial @Mock DataLoader loader;
```

4.8.5.5 Parameter and local variable annotations

There are no specific rules for formatting annotations on parameters or local variables (except, of course, when the annotation is a type-use annotation).

4.8.6 Comments

This section addresses *implementation comments*. Javadoc is addressed separately in Section 7, Javadoc.

Any line break may be preceded by arbitrary whitespace followed by an implementation comment. Such a comment renders the line non-blank.

4.8.6.1 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           /* Or you can
 * okay.            // is this.          * even do this. */
 */
```

Comments are not enclosed in boxes drawn with asterisks or other characters.

> **Tip:** When writing multi-line comments, use the `/* ... */` style if you want automatic code formatters to re-wrap the lines when necessary (paragraph-style). Most formatters don't re-wrap lines in `// ...` style comment blocks.

4.8.6.2 TODO comments

Use `TODO` comments for code that is temporary, a short-term solution, or good-enough but not perfect.

A `TODO` comment begins with the word `TODO` in all caps, a following colon, and a link to a resource that contains the context, ideally a bug reference. A bug reference is preferable because bugs are tracked and have follow-up comments. Follow this piece of context with an explanatory string introduced with a hyphen `-`.

The purpose is to have a consistent `TODO` format that can be searched to find out how to get more details.

```
// TODO: crbug.com/12345678 - Remove this after the 2047q4 compatibility winc
```

Avoid adding TODOs that refer to an individual or team as the context:

```
// TODO: @yourusername - File an issue and use a '*' for repetition.
```

If your `TODO` is of the form "At a future date do something" make sure that you either include a very specific date ("Fix by November 2005") or a very specific event ("Remove this code when all clients can handle XML responses.").

4.8.7 Modifiers

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

```
public protected private abstract default static final sealed non-sealed
  transient volatile synchronized native strictfp
```

Modifiers on `requires` module directives, when present, appear in the following order:

```
transitive static
```

4.8.8 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 `30000000001`.

4.8.9 Text Blocks

The opening `"""` of a text block is always on a new line. That line may either follow the same indentation rules as other constructs, or it may have no indentation at all (so it starts at the left margin). The closing `"""` is on a new line with the same indentation as the opening `"""`, and may be followed on the same line by further code. Each line of text in the text block is indented at least as much as the opening and closing `"""`. (If a line is indented further, then the string literal defined by the text block will have space at the start of that line.)

The contents of a text block may exceed the column limit.

5 Naming

5.1 Rules common to all identifiers

Identifiers use only ASCII letters and digits, and, in a small number of cases noted below, underscores. Thus each valid identifier name is matched by the regular expression `\w+`.

In Google Style, special prefixes or suffixes are **not** used. For example, these names are not Google Style: `name_`, `mName`, `s_name` and `kName`.

5.2 Rules by identifier type

5.2.1 Package and module names

Package and module names use only lowercase letters and digits (no underscores). Consecutive words are simply concatenated together. For example, `com.example.deepspace`, not `com.example.deepSpace` or `com.example.deep_space`.

5.2.2 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`).

There are no specific rules or even well-established conventions for naming annotation types.

A *test* class has a name that ends with `Test`, for example, `HashIntegrationTest`. If it covers a single class, its name is the name of that class plus `Test`, for example `HashImplTest`.

5.2.3 Method names

Method names are written in lowerCamelCase.

Method names are typically verbs or verb phrases. For example, `sendMessage` or `stop`.

Underscores may appear in JUnit *test* method names to separate logical components of the name, with *each* component written in lowerCamelCase, for example `transferMoney_deductsFromSource`. There is no One Correct Way to name test methods.

5.2.4 Constant names

Constant names use `UPPER_SNAKE_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. Examples include primitives, strings, immutable value classes, and anything set to `null`. If any of the instance's observable state can change, it is not a constant. Merely *intending* to never mutate the object is not enough. Examples:

```
// Constants
static final int NUMBER = 5;
static final ImmutableList<String> NAMES = ImmutableList.of("Ed", "Ann");
static final Map<String, Integer> AGES = ImmutableMap.of("Ed", 35, "Ann", 32)
static final Joiner COMMA_JOINER = Joiner.on(','); // because Joiner is immut
static final SomeMutableType[] EMPTY_ARRAY = {};

// Not constants
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(
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"};
```

These names are typically nouns or noun phrases.

5.2.5 Non-constant field names

Non-constant field names (static or otherwise) are written in lowerCamelCase.

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

5.2.6 Parameter names

Parameter names are written in lowerCamelCase.

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

5.2.7 Local variable names

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.

5.2.8 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 (see Section 5.2.2, Class names), followed by the capital letter `T` (examples: `RequestT`, `FooBarT`).

5.3 Camel case: defined

Sometimes there is more than one reasonable way to convert an English phrase into camel case, such as when acronyms or unusual constructs like "IPv6" or "iOS" are present. To improve predictability, Google Style specifies the following (nearly) deterministic scheme.

Beginning with the prose form of the name:

1. Convert the phrase to plain ASCII and remove any apostrophes. For example, "Müller's algorithm" might become "Muellers algorithm".
2. Divide this result into words, splitting on spaces and any remaining punctuation (typically hyphens).
    - *Recommended:* if any word already has a conventional camel-case appearance in common usage, split this into its constituent parts (e.g., "AdWords" becomes "ad words"). Note that a word such as "iOS" is not really in camel case *per se*; it defies *any* convention, so this recommendation does not apply.
3. Now lowercase *everything* (including acronyms), then uppercase only the first character

of:

   - ... each word, to yield *upper camel case*, or
   - ... each word except the first, to yield *lower camel case*
4. Finally, join all the words into a single identifier. Note that the casing of the original words is almost entirely disregarded.

In very rare circumstances (for example, multipart version numbers), you may need to use underscores to separate adjacent numbers, since numbers do not have upper and lower case variants.

Examples:

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` *	
"Turn on 2SV"	`turnOn2sv`	`turnOn2Sv`
"Guava 33.4.6"	`guava33_4_6`	`guava3346`

*Acceptable, but not recommended.

> **Note:** Some words are ambiguously hyphenated in the English language: for example "nonempty" and "non-empty" are both correct, so the method names `checkNonempty` and `checkNonEmpty` are likewise both correct.

6 Programming Practices

6.1 `@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, an interface method respecifying a superinterface method, and an explicitly declared accessor method for a record component.

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

6.2 Caught exceptions: not ignored

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);
```

6.3 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
```

6.4 Finalizers: not used

Do not override `Object.finalize`. Finalization support is *scheduled for removal*.

7 Javadoc

7.1 Formatting

7.1.1 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. */
```

The basic form is always acceptable. The single-line form may be substituted when the entirety

of the Javadoc block (including comment markers) can fit on a single line. Note that this only applies when there are no block tags such as `@param`.

7.1.2 Paragraphs

One blank line—that is, a line containing only the aligned leading asterisk (`*`)—appears between paragraphs, and before the group of block tags if present. Each paragraph except the first has `<p>` immediately before the first word, with no space after it. HTML tags for other block-level elements, such as `<ul>` or `<table>`, are *not* preceded with `<p>`.

7.1.3 Block tags

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

7.2 The summary fragment

Each Javadoc block 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...`, nor does it form a complete imperative sentence like `Save the record.`. However, the fragment is capitalized and punctuated as if it were a complete sentence.

> **Tip:** A common mistake is to write simple Javadoc in the form `/** @return the customer ID */`. This is incorrect, and should be changed to `/** Returns the customer ID. */` or `/** {@return the customer ID} */`.

7.3 Where Javadoc is used

At the *minimum*, Javadoc is present for every *visible* class, member, or record component, with a few exceptions noted below. A top-level class is visible if it is `public`; a member is visible if it is `public` or `protected` and its containing class is visible; and a record component is visible if its containing record is visible.

Additional Javadoc content may also be present, as explained in Section 7.3.4, Non-required Javadoc.

7.3.1 Exception: self-explanatory members

Javadoc is optional for "simple, obvious" members and record components, such as a `getFoo()` method, *if* there *really and truly* is nothing else worthwhile to say but "the foo".

> **Important:** it is not appropriate to cite this exception to justify omitting relevant information

> **Important:** It is not appropriate to use this exception to justify omitting relevant information that a typical reader might need to know. For example, for a record component named `canonicalName`, don't omit its documentation (with the rationale that it would say only `@param canonicalName the canonical name`) if a typical reader may have no idea what the term "canonical name" means!

7.3.2 Exception: overrides

Javadoc is not always present on a method that overrides a supertype method.

7.3.4 Non-required Javadoc

Other classes, members, and record components 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 `/**`).

Non-required Javadoc is not strictly required to follow the formatting rules of Sections 7.1.1, 7.1.2, 7.1.3, and 7.2, though it is of course recommended.