Java Coding Conventions
Post-AP Software Engineering
2023-2024

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.*

[1]

This document serves as the Post-AP Computer Science coding standards for source code written in the Java programming language. Java source code is considered standards compliant only if it adheres to these coding standards.

This document is created largely in conformance with *Java Code Conventions* [1] and the *Google Java Style Guide* [2], along with guidance from *Elements of Style* [3] and *The Elements of Java Style* [4]. This document does not cover all the elements of the Java programming language but does cover the elements most commonly used.

Contents

1. Source Files

1.1. File Name

The source file name consists of the case-sensitive name of the top-level (non-nested) class it contains (of which there is exactly one), plus the `.java` extension.

1.2. File Structure

A source file consists of, in order:

- License or copyright information if present
- Package statement
- Import statements
- Class/interface documentation comment
- Exactly one top-level (non-nested) class

1.3. Import Statements

Wildcard inputs, static or otherwise, are not used.

All static imports are in a single block, followed by a blank line, followed by all non-static imports in a single block. Each block is sorted alphabetically.

1.4. Class Structure

The contents of a class or interface should appear in the following order:

- Class (static) variables
- Instance variables
- Constructors
- Methods

Methods should appear in some form of a logical order. (Alphabetical or chronological are not logical orders.) Typically, higher-level methods appear before the lower-level methods they rely on. The goal is to use an order that makes reading and understanding (i.e., learning) the code easier.

All overloaded methods should appear in a single contiguous group, with no other methods in between.

2. Formatting

2.1. Indentation

Each time a new block or block-like construct is opened, the indent increases by four 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.

2.2. One Statement per Line

Each statement is followed by a line break.

2.3. Line Length

Avoid lines longer than 100 characters so that the entire line can be seen without horizontal scrolling.

2.4. Line Wrapping

When code that might otherwise legally occupy a single line is divided into multiple lines, this activity is called *line-wrapping*. Two considerations for line wrapping are where to break the line and where to start the next line.

Break according to these general principles:

- Break after a comma.
- Break before an operator.
- Prefer higher-level breaks to lower-level breaks.
- Keep a method attached to its open parenthesis.

Start the next line according to these general principles:

- Indent from the original line at least 8 spaces (two indentations).
- Align the new line with the beginning of the expression at the same level on the previous line.

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

Example line wrapping:

```
// Align on assignment operator
longName1 = longName2 * (longName3 + longName4
            - longName5) + 4 * longname6;

// Align on open parenthesis
someMethod(int anArg, Object anotherArg,
           String yetAnotherArg,
           Object andStillAnother) {
    ...
}

// Align on 8-space indendation
private synchronized horkingLongMethodName(int anArg,
        Object anotherArg, String yetAnotherArg,
        Object andStillAnother) {
    ...
}

// Align on open parenthesis
var = function1(longExpression1,
                function2(longExpression2,
                          longExpression3));

// Align on 8-space indendation
if ((condition1 && condition2)
        || (condition3 && condition4)
        ||!(condition5 && condition6)) {
    doSomethingAboutIt();
}
```

3. Comments

Java programs can have two kinds of comments: implementation comments and documentation comments.

3.1. Implementation Comments

Implementation comments should provide information related to the code that is not readily available in the code itself, such as information about nontrivial or nonobvious coding. Comments should contain only information that is relevant to reading and understanding the program. Comments should not duplicate information that is present in, and clear from, the code.

Add internal comments only if they will aid others in understanding your code. [4]

Describe why the code is doing what it does, not what the code is doing. [4]

Make sure comments and code agree. [3]

Don't comment bad code—rewrite it. [3]

3.1.1. Block Comments

Block comments are indented to the same level as the surrounding code (typically the code following the comment). Block comments can be single- or multi-line. Precede a block comment with a blank line.

Block comments can use the `/* ... */` style or the `// ...` style. For multi-line `/* ... */` comments, subsequent lines must start with `*` aligned with `*` on the previous line.

```
/*
 * This is
 * okay.
 */

// And so
// is this.

/* You can
 * also do this. */
```

3.1.2. End-line Comments

Short comments can appear on the same line as the code they describe. If an end-line comment causes the line to need to be wrapped, the comment should appear before the code.

3.2. Documentation Comments

Documentation comments are also known as Javadoc comments. Every `public` class, and every `public` and `protected` method within such a class, should have a Javadoc comment. The Javadoc comment should appear just before the declaration and should use the standard delimiters `/** ... */` with subsequent lines starting with `*` aligned with `*` on the previous line. The standard tags should appear in the order `@param`, `@return`, and `@throws`.

4. Declarations

4.1. Number per Line

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

4.2. Placement

Local variables are **not** habitually declared at the start of their containing block. Instead, local variables are declared close to the point they are first used (within reason), to minimize their scope.

4.3. Initialization

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

5. Statements

5.1. Simple Statements

Each line should contain at most one statement.

5.2. 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 to the beginning of the compound statement.
- Braces are used around all statements, even singletons, 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 forgetting to add braces.

5.3. *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();
```

5.4. 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;
}
```

5.5. *for* Statements

A `for` statement should have the following form:

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

for (type variable : collection) {
    statements;
}
```

5.6. *while* Statements

A while statement should have the following form:

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

5.7. *do-while* Statements

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

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

5.8. *switch* Statements

A `switch` statement should have the following form:

```
switch (input) {
    case 1:
    case 2:
        statements;
        // fall through
    case 3:
        statements;
        break;
    case 4:
        statements;
        break;
    default:
        statements;
        break;
}
```

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 `// fall 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.

5.9. *try-catch* Statements

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

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

6. Whitespace

6.1. Blank Lines

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

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 or single-line comment
- Between logical sections inside a method to improve readability

6.2. 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 (!done) {
       ...
  }
  ```

- Note that a blank space should not be used between a method name and its opening parenthesis. This helps to distinguish keywords from 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 (n < 10) {
    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. Examples:

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

7. Naming Conventions

Naming conventions make programs more understandable by making them easier to read. They can also give information about the function, or the purpose, of the identifier—for example, whether it's a constant, package, or class—which can be helpful in understanding the code.

Use variable names that mean something. [3]

Use familiar names. [4]

Choose variable names that won't be confused. [3]

Capitalize only the first letter in an acronym. [4]

The conventions given in this section are high level.

Identifier Type	**Rules for Naming**
Classes	Class names are typically nouns or noun phrases, in mixed case with the first letter of each internal word capitalized (UpperCamelCase). 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). Example: `public class Student`
Interfaces	Interface names should be capitalized like class names.

Methods	Methods are typically verbs or verb phrases, in mixed case with the first letter of each internal word capitalized (lowerCamelCase). Examples:
	
	`getArea();`
	`isEmpty();`

Variables	Except for class constants, all instance, class, parameter, and local variables are in mixed case with a lowercase first letter. Internal words start with capital letters (lowerCamelCase). Variable names should be short yet meaningful. The choice of a variable name should be 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.
Constants	The names of variables declared as class constants and should be all uppercase with words separated by underscores (UPPER_SNAKE_CASE). Example: `static final int MIN_WIDTH = 4;`

8. Style Guidelines

8.1. Algorithms

Write clearly – don't be too clever. [3]

Say what you mean, simply and directly. [3]

Write clearly – don't sacrifice clarity for efficiency. [3]

Make it right before you make it faster.

Make it fail-safe before you make it faster.

Make it clear before you make it faster.

Keep it simple to make it faster. [3]

8.2. Expressions

When working with expressions, make them understandable. This includes making good use of white space and line breaks, arranging the expression to reflect the logic, and using methods for repeated expressions. Also keep in mind that floating point expressions do not always calculate to the exact value we expect.

Parenthesize to avoid ambiguity. [3]

If a logical expression is hard to understand, try transforming it. [3]

Replace repeated nontrivial expressions with equivalent methods. [4]

10.0 times 0.1 is hardly ever 1.0. [3]

7/8 is zero while 7.0/8.0 is not zero. [3]

Don't compare floating point numbers solely for equality. [3]

8.3. Testing

Don't stop at one bug. [3]

Watch out for off-by-one errors. [3]

Make sure your code does "nothing" gracefully. [3]

Test programs at their boundary values. [3]

Check some answers by hand. [3]

8.4. User Interfaces

When your program interfaces with a user, the behavior of your program should not surprise the user.

Adhere to the Principle of Least Astonishment. [4]

Plan on the user not inputting what you expect, so test input and respond accordingly.

Test input for plausibility and validity. [3]

Make sure input doesn't violate the limits of the program. [3]

9. References

[1] Sun Microsystems, Inc., "Java Code Conventions," 20 April 1999. [Online]. Available: https://www.oracle.com/docs/tech/java/codeconventions.pdf. [Accessed 24 September 2022].

[2] Google LLC, "Google Java Style Guide," [Online]. Available: https://google.github.io/styleguide/javaguide.html. [Accessed 24 September 2022].

[3] B. W. Kernighan and P. J. Plauger, The Elements of Programming Style, 2nd ed., New York: McGraw Hill, 1978.

[4] A. Vermeulen, S. W. Ambler, G. Bumgardner, E. Metz, T. Misfeldt, J. Shur and P. Thompson, The Elements of Java Style, Cambridge: Cambridge University Press, 2000.

~~Write clearly – don't be too clever.~~

~~Say what you mean, simply and directly.~~

~~Use library functions whenever feasible.~~

Avoid too many temporary variables.

~~Write clearly – don't sacrifice clarity for efficiency.~~

Let the machine do the dirty work.

~~Replace repetitive expressions by calls to common functions.~~

~~Parenthesize to avoid ambiguity.~~

~~Choose variable names that won't be confused.~~

~~Avoid unnecessary branches.~~

~~If a logical expression is hard to understand, try transforming it.~~

~~Choose a data representation that makes the program simple.~~

Write first in easy-to-understand pseudo language; then translate into whatever language you have to use.

Modularize. Use procedures and functions.

~~Avoid gotos completely if you can keep the program readable.~~

Don't patch bad code – rewrite it.

Write and test a big program in small pieces.

Use recursive procedures for recursively-defined data structures.

Test input for plausibility and validity.

Make sure input doesn't violate the limits of the program.

~~Terminate input by end-of-file marker, not by count.~~

Identify bad input; recover if possible.

~~Make input easy to prepare and output self-explanatory.~~

~~Use uniform input formats.~~

~~Make input easy to proofread.~~

~~Use self-identifying input. Allow defaults. Echo both on output.~~

~~Make sure all variables are initialized before use.~~

Don't stop at one bug.

~~Use debugging compilers.~~

Watch out for off-by-one errors.

Take care to branch the right way on equality.

Be careful if a loop exits to the same place from the middle and the bottom.

Make sure your code does "nothing" gracefully.

Test programs at their boundary values.

Check some answers by hand.

~~10.0 times 0.1 is hardly ever 1.0.~~

~~7/8 is zero while 7.0/8.0 is not zero.~~

~~Don't compare floating point numbers solely for equality.~~

~~Make it right before you make it faster.~~

~~Make it fail-safe before you make it faster.~~

~~Make it clear before you make it faster.~~

~~Don't sacrifice clarity for small gains in efficiency.~~

~~Let your compiler do the simple optimizations.~~

Don't strain to re-use code; reorganize instead.

Make sure special cases are truly special.

~~Keep it simple to make it faster.~~

~~Don't diddle code to make it faster – find a better algorithm.~~

Instrument your programs. Measure before making efficiency changes.

~~Make sure comments and code agree.~~

~~Don't just echo the code with comments – make every comment count.~~

~~Don't comment bad code – rewrite it.~~

~~Use variable names that mean something.~~

~~Use statement labels that mean something.~~

Format a program to help the reader understand it.

~~Document your data layouts.~~

Don't over-comment