The following mild coding standard ensures easily readable and modifiable Java source code. This standard is based on [1, 2], where much more detailed rules, motivations, and examples can be found. The notes below explain these rules.

Consistency 1. Be consistent when using the freedom that this standard leaves.

Indentation 2. *Always* indent *systematically* a multiple of 4 spaces. *Never* use TAB characters in source code. (Let your editor expand the TAB key to spaces.)

Line length 3. *Always* limit the line length to at most 80 characters. (Set a right margin.)

One per line 4. *Always* write the following items on separate lines:

- variable declarations (including constants via `static final`)
- statements

That is, do not put two or more declarations or statements on the same line.

Empty lines 5. *Always* write one empty line before and after the following items:

- *group* of instance or local variable declarations
- each method and class declaration

Avoid spurious empty lines and groups of multiple empty lines.

Spacing 1 6. *Never* write a space before and *always* write one space after the following items (*unless* at line end):

- `,  ;`

Spacing 2 7. *Always* write one space before and after the following items (*unless* at line begin/end):

- keywords: `if  for  while` etc.
- binary operators (*except* `.`): `=  +  -  *  /  %  ==  !=  <  >  <=  >=  &&  ||` etc.

Curly braces 8. *Always* use curly braces `{ }` after

- `if (...)`
- `else`, *unless* immediately followed by `if` for multiway selection
- `while (...)`
- `for (...)`
- `do`
- `try` and `catch (...)`

Comments 9. *Always* explain each variable declaration in a comment.

Javadoc 10. *Always* specify each public entity in a javadoc comment.

BAD

```
public static int power(int a,int b) { // VIOLATES ALL RULES
    if  ( b<0)  throw new IllegalArgumentException ("power: negative
   int x=a,n=b ,r=1;
    while (n!=0)
      if(n%2==0){x=x*x;n= n/2; }

     else { r=r*x; n=n-1;}
        return r;
}
```

GOOD

```
/**
 * Returns {@code a} to the power {@code b}.
 *
 * @param   a  value of base
 * @param   b  value of exponent
 * @return  {@code a^b}, if {@code 0 <= b}
 * @throws  IllegalArgumentException  if {@code b < 0}
 */
public static int power(final int a, final int b) {
    if (b < 0) {
        throw new IllegalArgumentException(
                "power: exponent " + b + " is negative");
    }
    // 0 <= b

    int x = a; // see inv
    int n = b; // see inv
    int r = 1; // see inv

    // inv: 0 <= n <= b  &&  r * x^n == a^b
    while (n != 0) {
        if (n % 2 == 0) {  // even exponent
            x = x * x;
            n = n / 2;
        } else {  // odd exponent
            r = r * x;
            n = n - 1;
        }
        // inv holds again, n has decreased
    }
    // n == 0, hence r = a^b

    return r;
}
```

Notes

Well-organized source code is important for several reasons.

- The compiler may not care about this, but source code is also read by others: developers, reviewers, maintainers, teachers, graders, ...
- It is an important means to *prevent defects*.
- It facilitates *localization of defects*, both by the author, and by others.

Here is some further background information on each of the rules.

Consistency 1. Consistency especially plays a role in the placement of opening braces `{`. Either place them at the end of the line with the controlling statement, or at the beginning of a line by themselves directly below the the controlling statement. An advantage of the latter style is that opening and closing braces are vertically aligned. A disadvantage is that it takes more vertical space.

Indentation 2. Indentation provides visual clues about the containment structure (*nesting*). One or two space indentation does not provide enough visually guidance. Some standards prescribe indenting by multiples of three spaces (because that interferes with TAB characters, thereby discouraging them even more). Indenting by more than four spaces is a waste, and leaves less room in view of the line length limit (also see the next note).

Line length 3. Your screen may fit longer lines, but you are not the only one reading the source code. Moreover, long lines are hard to parse. Also see next note.

Avoid long lines by introducing auxiliary variables, methods, or classes (e.g., to group multiple parameters).

If a long line is unavoidable, break it at an appropriate place, and continue on the next line after a *double indentation*.

Of course, the line length of generated code may not be under your control.

One per line 4. See preceding note. Multiple statements on a single line make it harder:

- to add comments (see Rule 9);
- to move around code fragments;
- to interpret compiler messages;
- to set breakpoints precisely in a debugger.

Empty lines 5. Empty lines provide visual clues about *grouping*, on an intermediate level (also see next two notes). Besides the situations mentioned in Rule 5, it is good to delineate *groups of related statements* by empty lines; for example, initialization, loop, and finalization (cf. the example above). By the way, such grouping can also be made explicit by *curly braces*, defining a *block*, possibly with its own local variables.

Avoid long blocks of statements by introducing auxiliary methods.

Spacing 1

6. Spacing also provides visual clues about *grouping*, but on a lower level than empty lines (see preceding note; also see next note). This rule concerns *punctuation*.

   Commas are used to separate items in lists, such as *parameters* (both formal and actual), and expressions in an *array initializer*, but they also appear in **`throws`** and **`implements`** clauses.

   Multiple *semicolons* should only appear on one line to separate the three elements of a **`for`** statement.

Spacing 2

7. Spacing improves reqdability, especially when quickly scanning source code, rather than reading it slowly in full detail.

   Readability of expressions can be further improved by appropriate use of *parentheses*, and *auxiliary variables and functions*.

Curly braces

8. The Java syntax requires curly braces only when a statement[1] is to control more than one other statement. However, source code evolves, and statements get added and removed. To avoid spending any effort on deciding whether to include or remove braces, the rule is simply to write them always.

   But there is more to it. The controlled statement itself can span multiple lines of code, even if it is a single statement (e.g., an **`if`** or **`for`** statement). Proper indentation provides a visual clue where the controlled statement ends. But indentation is not taken into account by the compiler. Using braces ensures that there is both a visual clue and a clue that the compiler understands. Braces are thus a means to prevent defects that are otherwise hard to spot.

   Many source code editors are aware of braces, and these can help visualize, navigate, and fold source code.

   The rule makes one exception: *multiway selections*. To avoid *indentation creep*, a multiway selection with more than two branches uses **`else if`** as if it were a single keyword, with all the branches indented one level:

GOOD

```
if (x < lowerBound) {
    message = "too small";
} else if (upperBound < x) {
    message = "too large";
} else { // lowerBound <= x <= upperBound
    message = "just right";
}
```

[1]The **`switch`** statement is an exception: it syntactically requires braces.

and not

BAD

```
if (x < lowerBound) {
    message = "too small";
} else {
    if (upperBound < x) {
        message = "too large";
    } else { // lowerBound <= x <= upperBound
        message = "just right";
    }
}
```

Comments

9. Variables are introduced for a specific purpose. The name of the variable should reflect that purpose. However, a name should also not be too long. Furthermore, the purpose usually involves relationships to other elements of the program. A comment makes this explicit. To avoid having to spend any effort on deciding whether to include a comment or not, the rule is simply to provide it always.

   It is also good practice to provide comments with non-obvious statements. However, there is such a thing as *superfluous* comments. Do not comment the obvious.

Javadoc

10. Public entities are typically classes, interfaces, methods, and constants[2], but not instance variables. Public entities can be used anywhere in a program. Therefore, their usage should be well documented. Javadoc comments have two benefits over ordinary (non-javadoc) comments:

    - They support additional features, such as *tags*, to structure documentation.
    - They can be extracted from the source code and presented separately, as a document with cross references.

    Many Java Integrated Development Environments (IDEs) provide additional benefits when using javadoc comments.

References

[1] *Code Conventions for the Java Programming Language*. Sun, 1999. `java.sun.com/docs/codeconv`

[2] *Java Coding Standards*. ESA, 2005. `ftp.estec.esa.nl/pub/wm/wme/bssc/Java-Coding-Standards-20050303-releaseA.pdf`

[2]Constants are declared via **`public static final`**.