Java Coding Standard (Beginner/Intermediate Level)

Ada Lovlace, the first computer programmer

Naming Conventions

General Naming Conventions

1. Names representing packages should be in all lower case.

```
com.company.application.ui
```

The usual convention in package names is that the prefix should be the internet domain of the server which hosts the application, but in reverse order. The suffix depends on the application and the grouping. `e.g. com.microsoft.word.api, code.microsoft.word.recovery:` the first package will contain the API classes of word and the second package will contain the classes which handle the document recovery logic.

However, for your projects, the root name of the package should be your group name or project name followed by logical group names. `e.g. todobuddy.ui, todobuddy.file etc`.

> Note: Your code is not officially '*produced by NUS*', therefore do not use `edu.nus.comp.*` or anything similar.

2. Names representing classes or enum types must be nouns and written in mixed case starting with upper case.

```
Line, AudioSystem
```

Common practice in the Java development community and also the class naming convention used by Sun for the Java core packages.

3. Variable names must be in mixed case starting with lower case.

```
line, audioSystem
```

Common practice in the Java development community and also the naming convention for variables used by Sun for the Java core packages. Makes variables easy to distinguish from types, and effectively resolves potential naming collision as in the declaration `Line line;`

4. Names representing constants (final variables) or enum constants must be all uppercase using underscore to separate words.

```
MAX_ITERATIONS, COLOR_RED
```

Common practice in the Java development community and also the naming convention used by Sun for the Java core packages. You can do this easily in Eclipse by right clicking on the value you want to make a constant and selecting `Refactor->Extract Constant` .

5. Names representing methods must be verbs and written in mixed case starting with lower case.

```
getName(), computeTotalWidth()
```

Common practice in the Java development community and also the naming convention used by Sun for the Java core packages. This is identical to variable names, but methods in Java are already distinguishable from variables by their specific form.
Underscores may be used in test method names if your test method names are long and very descriptive. However, if this style is adopted for test methods, the whole team should follow it consistently.

`e.g. testLogic_addTask_nullParameters_errorMessageExpected()`

6. Abbreviations and acronyms should not be uppercase when used as a (OR part of a) name.

Good	Bad
`exportHtmlSource();`	`exportHTMLSource();`
`openDvdPlayer();`	`openDVDPlayer();`

Using all uppercase for the base name will give conflicts with the naming conventions given above. A variable of this type would have to be named `dVD` and `hTML` which is not very readable.

Another problem is illustrated in the examples above. When abbreviations/acronyms are used as part of a name, readability is seriously reduced. The word following the abbreviation/acronym does not stand out as it should.

As a result, minimize usage of abbreviations/acronyms, unless the concepts they are referring to are better known otherwise

> Note: A key point here is to use whatever that is easily understood by other people.

7. Private class variables can have underscore suffix or prefix (not common, but a beneficial practice)

```
class Person {
    private String name_; // OR private String _name;
    ...
}
```

Apart from its name and its type, the scope of a variable is its most important feature. Indicating class scope by using underscore makes it easy to distinguish class variables from local scratch variables. This is important because class variables are considered to have higher significance than method variables, and should be treated with special care by the programmer.

A side effect of the underscore naming convention is that it nicely resolves the problem of finding reasonable variable names for setter methods:

```
void setName(String name) {
    name_ = name;
}
```

8. All names should be written in English.

English is the preferred language for international development.

9. Variables with a large scope should have long names, variables with a small scope can have short names.

Scratch variables used for temporary storage or indices are best kept short. A programmer reading such variables should be able to assume that its value is not used outside a few lines of code. Common scratch variables for integers are `i, j, k, m, n` and for characters `c` and `d`.

Specific Naming Conventions

10. Boolean variables should be prefixed with 'is'

```
isSet, isVisible, isFinished, isFound, isOpen
```

This is the naming convention for boolean methods and variables used by Sun for the Java core packages.

Using the is prefix solves a common problem of choosing bad boolean names like *status* or *flag*. `isStatus` or `isFlag` simply doesn't fit, and the programmer is forced to chose more meaningful names.

Setter methods for boolean variables must have set prefix as in:

```
void setFound(boolean isFound);
```

There are a few alternatives to the is prefix that fits better in some situations. These are *has*, *can* and *should* prefixes:

```
boolean hasLicense();
boolean canEvaluate();
boolean shouldAbort = false;
```

> Note: Avoid boolean variables that represent the negation of a thing. e.g., use `isInitialized` instead of `isNotInitialized`

11. Plural form should be used on names representing a collection of objects.

```
Collection<Point> points;
int[] values;
```

Enhances readability since the name gives the user an immediate clue of the type of the variable and the operations that can be performed on its elements. One space character after the variable type is enough to obtain clarity.

12. Iterator variables should be called *i, j, k* etc.

```
for (Iterator i = points.iterator(); i.hasNext(); ) {
    ...
}

for (int i = 0; i < nTables; i++) {
    ...
}
```

The notation is taken from mathematics where it is an established convention for indicating iterators. Variables named *j, k* etc. should be used for nested loops only.

13. Associated constants (final variables) should be prefixed by a common type name.

```
final int COLOR_RED   = 1;
final int COLOR_GREEN = 2;
final int COLOR_BLUE  = 3;
```

This indicates that the constants belong together, and what concept the constants represents.

Files

1. Java source files should have the extension .java.

```
Point.java
```

Enforced by the Java tools.

2. Classes should be declared in individual files with the file name matching the class name. Secondary private classes can be declared as inner classes and reside in the file of the class they belong to.

```
class NusStudent extends Student {
    //Logic related to NusStudent class
    private class Module {
        //Logic related to Module class
    }
}
```

Enforced by the Java tools.

3. Use line wrapping to improve readability.

When wrapping lines, the main objective is to improve readability. Feel free to break rules if it improves readability. Do not always accept the auto-formatting suggested by the IDE.

Length: A line should be split if it exceeds the 110 characters. But it is OK to exceed the limit slightly or wrap the lines much shorter than limit.

Where to insert the break?

In general:

- Break after a comma.
- Align the new line with the beginning of the parent element.

```
method(param1, param2,
       param3, param4);
method(param1,
       method(param2,
              param3),
       param3);
```

- Break before an operator. This also applies to the following "*operator-like*" symbols: the dot separator `.`, the ampersand in type bounds `<T extends Foo & Bar>`, and the pipe in catch blocks `catch (FooException | BarException e)`

```
totalSum = a + b + c
         + d + e;
setText("Long line split"
        + "into two parts.");
method(param1,
       object.method()
```

```
            .method2(),
        param3);
```

- A method or constructor name stays attached to the open parenthesis `(` that follows it.

Good

```
someMethodWithVeryVeryVeryVeryVeryVeryVeryVeryVeryVeryVeryLongName(
        int anArg, Object anotherArg);
```

Bad

```
someMethodWithVeryVeryVeryVeryVeryVeryVeryVeryVeryVeryVeryLongName
        (int anArg, Object anotherArg);
```

- Prefer higher-level breaks to lower-level breaks. In the example below, the first is preferred, since the break occurs outside the parenthesized expression, which is at a higher level.

Good

```
// PREFER THIS
longName1 = longName2 * (longName3 + longName4 - longName5)
            + 4 * longname6;
```

Bad

```
// OVER THIS
longName1 = longName2 * (longName3 + longName4
            - longName5) + 4 * longname6;
```

- Single-column stacking of parameters or exceptions is discouraged in most cases, unless the column is wide enough. While such stacking improves the list of parameters/exceptions, it may not outweigh the cost of increased height of the code.

Good

```
//BECAUSE THE COLUMN IS WIDE
longMethod(someLongMenthod1(param1, param2, param3).anotherMethod(),
           someLongMenthod2(param1, param2).anotherMethod(),
           someLongMenthod3(param1, param2, param3));
```

Bad

```
method(param1,
       param2,
       param3,
       param4);
void method(param1,param2)throws Exception1,
                                 Exception2,
                                 Exception3 {
```

- Here are three acceptable ways to format ternary expressions:

```
alpha = (aLongBooleanExpression) ? beta : gamma;
alpha = (aLongBooleanExpression) ? beta
                                 : gamma;
alpha = (aLongBooleanExpression)
      ? beta
      : gamma;
```

- If the above rules lead to confusing code or to code that's squished up against the right margin, just indent 8 spaces instead.

Good

```
//INDENT 8 SPACES TO AVOID VERY DEEP INDENTS
private static synchronized horkingLongMethodName(int arg, Object anotherArg,
        String yetAnotherArg, Object andStillAnother) {
    ...
}
```

```
//EVEN THIS IS ACCEPTABLE (2nd line starts to the right of method name)
private static synchronized horkingLongMethodName(int arg, Object anotherArg,
                                String yetAnotherArg, Object andStillAnother) {
    ...
}
```

Bad

```
//CONVENTIONAL INDENTATION (code getting squished up against the right)
void someMethodWithVeryVeryVeryVeryVeryVeryLongName(int arg,
                                                    Object anotherArg,
                                                    String yetAnotherArg,
                                                    Object andStillAnother) {
    ...
}
```

Statements

Package and Import Statements

1. The package statement must be the first statement of the file. All files should belong to a specific package.

The package statement location is enforced by the Java language. A Java package is a set of classes which are grouped together. Every class is part of some package. You can use packages to organise your code. It will help you and other developers easily understand the code base when all the classes have been grouped in packages.

The rule of thumb is to package the classes that are related. For example in Java, the classes related to file writing is grouped in the package `java.io` and the classes which handle lists, maps etc are grouped in `java.util` package.

2. The import statements must follow the package statement. import statements should be sorted with the most fundamental packages first, and grouped with associated packages together and one blank line between groups.

```
import java.io.IOException;
import java.net.URL;
import java.rmi.RmiServer;
import java.rmi.server.Server;
import javax.swing.JPanel;
import javax.swing.event.ActionEvent;
import org.linux.apache.server.SoapServer;
```

The import statement location is enforced by the Java language. The sorting makes it simple to browse the list when there are many imports, and it makes it easy to determine the dependencies of the present package. The grouping reduce complexity by collapsing related information into a common unit.

> *Hint: You can organise the imports automatically by simply pressing CTRL+SHIFT+O in Eclipse.*

3. Imported classes should always be listed explicitly.

Good **Bad**

```
import java.util.List;
import java.util.ArrayList;
import java.util.HashSet;
```

```
import java.util.*;
```

Importing classes explicitly gives an excellent documentation value for the class at hand and makes the class easier to comprehend and maintain. Appropriate tools should be used in order to always keep the import list minimal and up to date. For example, Eclipse IDE can do this easily.

Classes and Interfaces

4. Class and Interface declarations should be organized in the following manner:

1. Class/Interface documentation (Comments)
2. **class** or **interface** statement
3. Class (static) variables in the order **public**, **protected**, **package** (no access modifier), **private**
4. Instance variables in the order **public**, **protected**, **package** (no access modifier), **private**
5. Constructors
6. Methods (no specific order)

Make code easy to navigate by making the location of each class element predictable.
Methods

5. Method modifiers should be given in the following order:

```
<access> static abstract synchronized <unusual> final native
```

The <access> modifier (if present) must be the first modifier.

Good	Bad
`public static double square(double a);`	`static public double square(double a);`

```
<access> = public | protected | private
<unusual> = volatile | transient
```

The most important lesson here is to keep the *access* modifier as the first modifier. Of the possible modifiers, this is by far the most important, and it must stand out in the method declaration. For the other modifiers, the order is less important, but it make sense to have a fixed convention.

Types

6. Array specifiers must be attached to the type not the variable.

Good	Bad
`int[] a = new int[20];`	`int a[] = new int[20];`

The *arrayness* is a feature of the base type, not the variable. Sun allows both forms however.

Variables

7. Variables should be initialized where they are declared and they should be declared in the smallest scope possible.

Good

```
int sum = 0;
for (int i = 0; i < 10; i++) {
    for (int j = 0; j < 10; j++) {
        sum += i * j;
    }
}
```

Bad

```
int i, j, sum;
sum = 0;
for (i = 0; i < 10; i++) {
    for (j = 0; j < 10; j++) {
        sum += i * j;
    }
}
```

This ensures that variables are valid at any time. Sometimes it is impossible to initialize a variable to a valid value where it is declared. In these cases it should be left uninitialized rather than initialized to some phony value.

8. Class variables should never be declared public.

The concept of Java information hiding and encapsulation is violated by public variables. Use private variables and access functions instead. One exception to this rule is when the class is essentially a data structure, with no behavior (*equivalent to a C++ struct*). In this case it is appropriate to make the class' instance variables public.

Loops

9. The loop body should be wrapped by curly brackets irrespective of how many lines there are in the body

Good

```
sum = 0;
for (i = 0; i < 100; i++) {
    sum += value[i];
}
```

Bad

```
for (i = 0, sum = 0; i < 100; i++)
    sum += value[i];
```

When there is only one statement in the loop body it can be written without wrapping it between { }, however that is error prone and *very* strongly discouraged from using.

Conditionals

10. The conditional should be put on a separate line.

Good

```
if (isDone) {
    doCleanup();
```

Bad

```
if (isDone) doCleanup();
```

```
}
```

This is for debugging purposes. When writing on a single line, it is not apparent whether the test is really true or not.

11. Single statement conditionals should still be wrapped by curly brackets

Good

```
InputStream stream = File.open(fileName, "w");
if (stream != null) {
    readFile(stream);
}
```

Bad

```
InputStream stream = File.open(fileName, "w");
if (stream != null))
    readFile(stream);
```

The body of the conditional should be wrapped by curly brackets irrespective of how many statements are in it to avoid error prone code.

Layout and Comments

> Note: Many of the layout rules mentioned below can be applied in Eclipse by simply pressing *CTRL+SHIFT+F* (F for Format). If you want to format only a specific part of the code instead of the whole class, highlight the lines you want to format and then press *CTRL+SHIFT+F*. The Eclipse formatter will **not** work properly if you have intentionally pressed *enter/tab/space* keys in unwanted places. Therefore, even if you do use the Eclipse formatter, we highly recommend that you **double check** whether your code is in accordance with the rules mentioned below.
>
> Note: Another option is to use the Correct Indentation function found in Eclipse. This can be accessed by clicking `Source -> Correct Indentation (Ctrl-I)`. This function will merely indent the code instead of formatting it as compared with the Format function above.

Layout

1. Basic indentation should be 4 spaces.

```
for (i = 0; i < nElements; i++) {
    a[i] = 0;
}
```

Indentation is used to emphasize the logical structure of the code. Use 4 spaces to indent (not tabs).

Place the line breaks to improve readability. It is ok to exceed 110 char limit for a line, but not by too much. You can configure Eclipse to break lines after 110 chars, but sometimes the automatic line break does not give readability. In such cases, you can decide where to put the line break to have best readability.

2. Block layout should be as illustrated as shown below.

Good

```
while (!done) {
    doSomething();
    done = moreToDo();
}
```

Bad

```
while (!done)
    {
        doSomething();
        done = moreToDo();
    }
```

The Bad example introduces an extra indentation level which doesn't emphasize the logical structure of the code as clearly as the Good example.

3. Method definitions should have the following form:

```
public void someMethod() throws SomeException {
    ...
}
```

4. 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. The *for* statement should have the following form:

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

This follows from the general block rule above.

6. The *while* statement should have the following form:

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

This follows from the general block rule above.

7. The *do-while* statement should have the following form:

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

This follows from the general block rule above.

8. The *switch* statement should have the following form:

```
switch (condition) {
case ABC:
    statements;
    // Fallthrough
case DEF:
    statements;
    break;
case XYZ:
    statements;
    break;
default:
    statements;
    break;
}
```

The explicit `//Fallthrough` comment should be included whenever there is a `case` statement without a break statement. Leaving out the `break` is a common error, and it must be made clear that it is intentional when it is not there.

9. A *try-catch* statement should have the following form:

```
try {
    statements;
} catch (Exception exception) {
    statements;
}

try {
    statements;
} catch (Exception exception) {
    statements;
} finally {
    statements;
}
```

This follows partly from the general block rule above. This form differs from the Sun recommendation in the same way as the `if-else` statement described above.

White Space

10. Take note of the following:

- **Operators should be surrounded by a space character.**
- **Java reserved words should be followed by a white space.**
- **Commas should be followed by a white space.**
- **Colons should be surrounded by white space.**
- **Semicolons in for statements should be followed by a space character.**

Good	Bad
`a = (b + c) * d;`	`a=(b+c)*d;`
`while (true) {`	`while(true){`
`doSomething(a, b, c, d);`	`doSomething(a,b,c,d);`
`case 100 :`	`case 100:`
`for (i = 0; i < 10; i++) {`	`for(i=0;i<10;i++){`

Makes the individual components of the statements stand out and enhances readability. It is difficult to give a complete list of the suggested use of whitespace in Java code. The examples above however should give a general idea of the intentions.

11. Logical units within a block should be separated by one blank line.

```
// Create a new identity matrix
Matrix4x4 matrix = new Matrix4x4();

// Precompute angles for efficiency
double cosAngle = Math.cos(angle);
double sinAngle = Math.sin(angle);

// Specify matrix as a rotation transformation
matrix.setElement(1, 1,  cosAngle);
matrix.setElement(1, 2,  sinAngle);
matrix.setElement(2, 1, -sinAngle);
matrix.setElement(2, 2,  cosAngle);

// Apply rotation
transformation.multiply(matrix);
```

Enhances readability by introducing white space between logical units. Each block is often introduced by a comment as indicated in the example above.

Comments

12. Write minimal but sufficient comments.

In general, the use of comments should be minimized by making the code self-documenting by appropriate name choices and an explicit logical structure.

However, you MUST write header comments for all classes, public methods, and all non-trivial private methods. The code, even if it is self-explanatory, can only tell the reader HOW the code works, not WHAT the code is supposed to do.

13. All comments should be written in English.

In an international environment English is the preferred language.

14. Javadoc comments should have the following form:

```
/**
 * Return lateral location of the specified position.
 * If the position is unset, NaN is returned.
 *
 * @param x    X coordinate of position.
 * @param y    Y coordinate of position.
 * @param zone Zone of position.
 * @return     Lateral location.
 * @throws IllegalArgumentException  If zone is <= 0.
 */
public double computeLocation(double x, double y, int zone)
    throws IllegalArgumentException {
```

```
  ...
}
```

A readable form is important because this type of documentation is typically read more often inside the code than it is as processed text.

Note in particular:

- The opening `/**` on a separate line
- Write the first sentence as a short summary of the method, as Javadoc automatically places it in the method summary table (and index). See here (from [5]) for more info.
- Subsequent `*` is aligned with the first one
- Space after each `*`
- Empty line between description and parameter section
- Alignment of parameter descriptions
- Punctuation behind each parameter description
  -No blank line between the documentation block and the method/class

Javadoc of class members can be specified on a single line as follows:

```
/** Number of connections to this database */
private int nConnections;
```

15. Comments should be indented relative to their position in the code.

Good	Bad	Bad

```
while (true) {      while (true) {          while (true) {
    // Do something         // Do something // Do something
    something();        something();             something();
}                   }                       }
```

This is to avoid the comments from breaking the logical structure of the program.

References

[1] - http://geosoft.no/development/javastyle.html

[2] - http://www.oracle.com/technetwork/java/codeconventions-150003.pdf

[3] - http://developers.sun.com/sunstudio/products/archive/whitepapers/java-style.pdf

[4] - Effective Java, 2nd Edition by Joshua Bloch

[5] - http://www.oracle.com/technetwork/java/javase/documentation/index-137868.html

Prepared by: Nimantha Baranasuriya

Contributions: Dai Thanh, Tong Chun Kit

Converted from Google Docs to Markdown Document by: Barnabas Tan