Guides for SE student projects »

Java coding standard (basic + intermediate)

Versions: [Basic Rules] [Basic + Intermediate Rules] [All Rules]

💡 Use the Google Java style guide for any topics not covered in this document.

> **Legend**: ⭐ basic rule | ⭐⭐ intermediate rule | ⭐⭐⭐ advanced rule

- **Naming**
- **Layout**
- **Statements**: Package/Import | Types | Variables | Loops | Conditionals
- **Comments**
- References
- Contributors

Naming

⭐ **Names representing packages should be in all lower case.**

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

ℹ️ More on package naming

For school 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`.

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

⭐ **Class/enum names must be nouns and written in PascalCase.**

```
Line, AudioSystem
```

⭐ **Variable names must be in camelCase.**

```
line, audioSystem
```

⭐ **Constant** names must be all **uppercase** using **underscore** to separate words.

```
MAX_ITERATIONS, COLOR_RED
```

⭐ **Names representing methods** must be **verbs** and written in **camelCase.**

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

Underscores may be used in **test** method names using the following three part format
`featureUnderTest_testScenario_expectedBehavior()`

e.g. `sortList_emptyList_exceptionThrown()` `getMember_memberNotFount_nullReturned`

Third part or both second and third parts can be omitted depending on what's covered in the test. For example, the test method `sortList_emptyList()` will test `sortList()` method for all variations of the 'empty list' scenario and the test method `sortList()` will test the `sortList()` method for all scenarios.

⭐⭐ **Abbreviations** and **acronyms** should **not be uppercase** when used as a part of a name.

👍 **Good**

```
exportHtmlSource();
openDvdPlayer();
```

👎 **Bad**

```
exportHTMLSource();
openDVDPlayer();
```

⭐ **All names should be written in English.**

ℹ Rationale: The code is meant for an international audience.

⭐⭐ **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 can be 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`.

ℹ Rationale: When the scope is small, the reader does not have to remember it for long.

⭐ **Boolean variables/methods** should be named to **sound like booleans**

```
//variables
isSet, isVisible, isFinished, isFound, isOpen, hasData, wasOpen

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

> **i** As much as possible, use a prefix such as `is`, `has`, `was`, etc. for boolean variable/method names so that linters can automatically verify that this style rule is being followed.

Setter methods for boolean variables must be of the form:

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

ℹ Rationale: This is the naming convention for boolean methods and variables used by Java core packages. It also makes the code read like normal English e.g. `if(isOpen) ...`

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

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

ℹ Rationale: 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.

⭐ **Iterator variables can be called *i, j, k* etc.**

Variables named *j, k* etc. should be used for nested loops only.

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

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

ℹ Rationale: The notation is taken from mathematics where it is an established convention for indicating iterators.

⭐⭐ **Associated constants should have a common prefix.**

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

ℹ Rationale: This indicates that they belong together, and make them appear together when sorted alphabetically.

Layout

⭐ **Basic indentation should be 4 spaces (not tabs).**

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

ℹ️ Rationale: Just follow it 😛

⭐ **Line length should be no longer than 120 chars.**

Try to keep line length shorter than 110 characters (soft limit). But it is OK to exceed the limit slightly (hard limit: 120 chars). If the line exceeds the limit, use line wrapping at appropriate places of the line.

Indentation for wrapped lines should be 8 spaces (i.e. twice the normal indentation of 4 spaces) more than the parent line.

```
setText("Long line split"
        + "into two parts.");
if (isReady) {
    setText("Long line split"
            + "into two parts.");
}
```

⭐⭐ **Place line break to improve readability**

When wrapping lines, the main objective is to improve readability. Do not always accept the auto-formatting suggested by the IDE.

In general:

- Break after a comma.

- 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**
```
longName1 = longName2 * (longName3 + longName4 - longName5)
          + 4 * longname6
```

👎 **Bad**
```
longName1 = longName2 * (longName3 + longName4
          - longName5) + 4 * longname6;
```

- Here are two acceptable ways to format ternary expressions:

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

⭐ **Use K&R style brackets (aka Egyptian style).**

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

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

ⓘ Rationale: Just follow it. 😛

⭐ **Method definitions should have the following form:**

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

⭐ **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;
}
```

☆ The *for* statement should have the following form:

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

☆ The *while* and the *do-while* statements should have the following form:

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

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

☆ The *switch* statement should have the following form: Note there is no indentation for `case` clauses.

💡 Configure your IDE to follow this style instead.

```
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.

ⓘ Rationale: Leaving out the `break` is a common error, and it must be made clear that it is intentional when it is not there.

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

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

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

☆☆ **White space within a statement**

It is difficult to give a complete list of the suggested use of whitespace in Java code. The examples below however should give a general idea of the intentions.

Rule	👍 Good	👎 Bad
Operators should be surrounded by a space character.	`a = (b + c) * d;`	`a=(b+c)*d;`
Java reserved words should be followed by a white space.	`while (true) {`	`while(true){`
Commas should be followed by a white space.	`doSomething(a, b, c, d);`	`doSomething(a,b,c,d);`
Colons should be surrounded by white space when used as a binary/ternary operator. Does not apply to `switch x:`. Semicolons in `for` statements should be followed by a space character.	`for (i = 0; i < 10; i++) {`	`for(i=0;i<10;i++){`

ⓘ Rationale: Makes the individual components of the statements stand out and enhances readability.

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

```java
// 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);
```

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

Statements

Package and Import Statements

⭐ Put **every class in a package.**

Every class should be part of some package.

ⓘ Rationale: It will help you and other developers easily understand the code base when all the classes have been grouped in packages.

⭐⭐ The **ordering of import statements** must be **consistent.**

ⓘ Rationale: A consistent ordering of import statements makes it easier to browse the list and determine the dependencies when there are many imports.

Example:

```
import static org.junit.Assert.assertEquals;
import static org.junit.Assert.assertTrue;

import java.io.File;
import java.io.IOException;

import javax.xml.bind.JAXBContext;
import javax.xml.bind.JAXBException;

import org.loadui.testfx.GuiTest;
import org.testfx.api.FxToolkit;

import com.google.common.io.Files;

import javafx.geometry.Bounds;
import javafx.geometry.Point2D;
import junit.framework.AssertionFailedError;
```

💡 IDEs have support for auto-ordering import statements. However, note that the default orderings of different IDEs are not always the same. It is recommended that you and your team use the same IDE and stick to a consistent ordering.

⭐ **Imported classes should always be listed explicitly.**

👍 **Good**   👎 **Bad**

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

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

ℹ️ Rationale: 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. IDE's can be configured to do this easily.

Types

⭐ **Array specifiers must be attached to the type not the variable.**

👍 **Good**   👎 **Bad**

```java
int[] a = new int[20];
```

```java
int a[] = new int[20];
```

ℹ️ Rationale: The *arrayness* is a feature of the base type, not the variable. Java allows both forms however.

Variables

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

👍 **Good**   👎 **Bad**

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

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

ℹ️ Rationale: 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.

⭐⭐ **Class variables should never be declared public** unless the class is a *data class* with no behavior. This rule does not apply to constants.

👎 **Bad**

```java
public class Foo{

    public int bar;

}
```

ℹ️ Rationale: The concept of Java information hiding and encapsulation is violated by public variables. Use non-public variables and access functions instead.

Loops

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

👍 **Good**

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

👎 **Bad**

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

ℹ️ Rationale: When there is only one statement in the loop body, Java allows it to be written without wrapping it between `{ }`. However that is error prone and *very* strongly discouraged from using.

Conditionals

☆ **The conditional should be put on a separate line.**

👍 **Good**

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

👎 **Bad**

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

ℹ️ Rationale: This helps when debugging using an IDE debugger. When writing on a single line, it is not apparent whether the condition is really true or not.

⭐ **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.

ℹ️ Rationale: Omitting braces can lead to subtle bugs.

Comments

⭐ **All comments should be written in English.**

Furthermore, use American spelling and avoid local slang.

ℹ️ Rationale: The code is meant for an international audience.

⭐⭐ **Write descriptive header comments for all public classes/methods.**

You MUST write header comments for all classes, public methods. But they *can* be omitted for the following cases:
i. Getters/setters
ii. When overriding methods (provided the parent method's Javadoc applies exactly *as is* to the overridden method)

ℹ️ Rationale: `public` method are meant to be used by others and the users should not be forced to read the code of the method to understand its exact behavior. 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.

⭐ **Javadoc comments should have the following form:**

```java
/**
 * Returns 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 {
    //...
}
```

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).
  - In method header comments, the first sentence should start in the form `Returns ...`, `Sends ...`, `Adds ...` (not `Return` or `Returnning` etc.)
- Subsequent `*` is aligned with the first one
- Space after each `*`
- Empty line between description and parameter section
- Punctuation behind each parameter description
- No blank line between the documentation block and the method/class
- `@return` can be omitted if the method does not return anything
- When writing Javadocs for overridden methods, the `@inheritDoc` tag can be used to reuse the header comment from the parent method but with further modifications e.g., when the method has a slightly different behavior from the parent method.

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

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

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

👍 **Good**

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

👎 **Bad**

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

👎 **Bad**

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

ℹ Rationale: 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

Contributors

- Nimantha Baranasuriya - Initial draft
- Dai Thanh - Further tweaks
- Tong Chun Kit - Further tweaks
- Barnabas Tan - Converted from Google Docs to Markdown Document