Java Naming Convention

A Good Practice

Why to have naming convention

- 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.

Standard Guideline To Code

- Class/Interface- UpperCamelCase
- Variable/Methods- lowerCamelCase
- Constants- UPPERCASE
- Packages- lowercase

Identifier Type	Rules for Naming	Example
Calsses	Class names should be nouns, in mixed case with the first letter of each internal word capitalized. 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).	class Raster; class ImageSprite;
Intefaces	Interface names should be capitalized like class names	interface RasterDelegate; interface Storing;

Methods	Methods should be verbs, in mixed case with the first letter lowercase, with the first letter of each internal word capitalized.	run();
		runFast();
		getBackground();

Variable	Except for variables, all instance, class, and class constants are in mixed case with a lowercase first letter. Internal words start with capital letter	int i; char *cp; float myWidth;

Constant	The names of variables declared class constants and of ANSI constants should be all uppercase with words separated by underscores (“_”). (ANSI constants should be avoided, for ease of debugging.)	int MIN_WIDTH = 4;
		int MAX_WIDTH = 999;
		int GET_THE_CPU = 1;

File Names

- Java source should be suffixed with **.java** and Java bytecode **.class**
- **GNUmakefile:** The preferred name for makefiles. We use gnumake to build our software.
- **README:** The preferred name for the file that summarizes the contents of a particular directory.
- Files longer than 2000 lines are cumbersome and should be avoided
- A file consists of sections that should be separated by blank lines and an optional comment identifying each section.

Indentation and Line Length

- Four spaces should be used as the unit of indentation. The exact construction of the indentation (spaces vs. tabs) is unspecified.
- Tabs must be set exactly every 8 spaces (not 4).
- Avoid lines longer than 80 characters, since they’re not handled well by many terminals and tools. generally no more than 70 characters.

Java Source Files

- Each Java source file contains a single public class or interface.
- When private classes and interfaces are associated with a public class, you can put them in the same source file as the public class.
- The public class should be the first class or interface in the file
- Package and Import statements; for example:

```
import java.applet.Applet;
import java.awt.*;
import java.net.*;
Class and interface declarations
```

Comments

- All source files should begin with a **c-style comment** that lists the programmer(s), the date, a copyright notice, and also a brief description of the purpose of the program

```
/*
 * Classname
*
 * Version info
 *
 * Copyright notice
 */
```

Types Of Comments

- Document comment delimited by /*...*/, and implementation comment is //.
- Documentation comments (known as “doc comments”) are Java-only, and are delimited by /**...*/. Doc comments can be extracted to HTML files using the javadoc tool.
- Implementation comments used for commenting out code or for comments about the particular implementation.
- Doc comments are meant to describe the specification of the code, from an implementation-free perspective. to be read by developers who might not necessarily have the source code at hand.

Implementation Comment Formats

- Programs can have four styles of implementation comments: block, single-line, trailing and end-of-line

Implementation Block Comment

- Block comments are used to provide descriptions of files, methods, data structures and algorithms.
- Block comments should be used at the beginning of each file and before each method.
- They can also be used in other places, such as within methods. Block comments inside a function or method should be indented to the same level as the code they describe.
- A block comment should be preceded by a blank line to set it apart from the rest of the code.
- Block comments have an asterisk **“*”** at the beginning of each line except the first.

```
/*
 * Here is a block comment.
 */
```

Single Line Comment

- Short comments can appear on a single line indented to the level of the code that follows. If a comment can’t be written in a single line, it should follow the block comment format
- A single-line comment should be preceded by a blank line.

```
if (condition) {
 /* Handle the condition. */
 ...
}
```

Trailing Comment

- Very short comments can appear on the same line as the code they describe, but should be shifted far enough to separate them from the statements.
- If more than one short comment appears in a chunk of code, they should all be indented to the same tab setting.
- Avoid the assembly language style of commenting every line of executable code with a trailing comment

```
if (a == 2) {
 return TRUE; /* special case */
} else {
 return isprime(a); /* works only for odd a */
}
```

End-Of-Line Comment

- The // comment delimiter begins a comment that continues to the newline.
- It can comment out a complete line or only a partial line.
- It shouldn’t be used on consecutive multiple lines for text comments; however, it can be used in consecutive multiple lines for commenting out sections of code.

End-Of-Line Comment

```
if (foo > 1) {
    // Do a double-flip.
    ...
}
else
    return false; // Explain why here.
//if (bar > 1) {
//
//      // Do a triple-flip.
//      ...
//}
//else
//      return false;
```

Documentation Comment

- Doc comments describe Java classes, interfaces, constructors, methods, and fields.
- Each doc comment is set inside the comment delimiters /**...*/, with one comment per API.
- This comment should appear just before the declaration:

```
/**
 * The Example class provides ...
 */
class Example { ...
```

Package and Import Statements

- The first non-comment line of most Java source files is a package statement. After that, import statements can follow.

```
package java.awt;

import java.awt.peer.CanvasPeer;
```

Class/Interface Declaration	***Notes***
Class/interface implementation comment (/*...*/), if necessary	This comment should contain any class-wide or interface-wide information that wasn't appropriate for the class/interface documentation comment
Class (static) variables	First the public class variables, then the protected, and then the private4
Instance variables	First public, then protected, and then private.
Methods	These methods should be grouped by functionality rather than by scope or accessibility. For example, a private class method can be in between two public instance methods. The goal is to make reading and understanding the code easier.

Wrapping Lines

When an expression will not fit on a single line, break it according to these general principles:

- Break after a comma.
- Break before an operator.
- Prefer higher-level breaks to lower-level breaks.
- Align the new line with the beginning of the expression at the same level on the previous line.
- If the above rules lead to confusing code or to code that’s squished up against the right margin, just indent 8 spaces instead.

Wrapping Lines

```
function(longExpression1, longExpression2, longExpression3,
        longExpression4, longExpression5);
var = function1(longExpression1,
            function2(longExpression2,
            longExpression3));
longName1 = longName2 * (longName3 + longName4 - longName5)
                + 4 * longname6; // PREFER
longName1 = longName2 * (longName3 + longName4
                                - longName5) + 4 * longname6; // AVOID
```

Wrapping Lines

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

Wrapping Lines

```
//DON'T USE THIS INDENTATION
if ((condition1 && condition2)
    || (condition3 && condition4)
    ||!(condition5 && condition6)) { //BAD WRAPS
    doSomethingAboutIt(); //MAKE THIS LINE EASY TO MISS
}
//USE THIS INDENTATION INSTEAD
if ((condition1 && condition2)
            || (condition3 && condition4)
            ||!(condition5 && condition6)) {
    doSomethingAboutIt();
}
//OR USE THIS
if ((condition1 && condition2) || (condition3 && condition4)
            ||!(condition5 && condition6)) {
    doSomethingAboutIt();
}
```

Wrapping Lines

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

Variable Initialization

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

Class and Interface Declarations

- No space between a method name and the parenthesis “(“ starting its parameter list Open brace “{” appears at the end of the same line as the declaration statement Closing brace “}” starts a line by itself indented to match its corresponding opening statement, except when it is a null statement the “}” should appear immediately after the

```
“{“
class Sample extends Object {
        int ivar1;
        int ivar2;
        Sample(int i, int j) {
                ivar1 = i;
                ivar2 = j;
         }
        int emptyMethod() {}
        ...
}
```

- Methods are separated by a blank line

Simple Statements

- Each line should contain at most one statement.

```
argv++; argc--; // AVOID!
```

- Do not use the comma operator to group multiple statements unless it is for an obvious reason.

```
if (err) {
    Format.print(System.out, “error”), exit(1); //VERY WRONG!
}
```

Compound Statements

- A lists of statements enclosed in braces **“{ statements }”**.
- 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 a if-else or for statement.
- This makes it easier to add statements without accidentally introducing bugs due to forgetting to add braces.

Declarations: Number Per Line

- One declaration per line is recommended since it encourages commenting.

```
int level; // indentation level
int size; // size of table
```

- is preferred over `int level, size;`
- In absolutely no case should variables and functions be declared on the same line.

```
long dbaddr, getDbaddr(); // WRONG!
```

- Do not put different types on the same line.

```
int foo, fooarray[]; //WRONG!
```

- one space between the type and the identifier.

Declarations: Placement

- Put declarations only at the beginning of blocks. (A block is any code surrounded by curly braces “{” and “}”.)
- Don’t wait to declare variables until their first use; it can confuse the unwary programmer and hamper code portability within the scope.

```
void MyMethod() {
 int int1; // beginning of method block
 if (condition) {
 int int2; // beginning of "if" block
 ...
 }
}
```

- The one exception to the rule is indexes of for loops, which in Java can be declared in the for statement:

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

- Avoid local declarations that hide declarations at higher levels.
- do not declare the same variable name in an inner block

return Statements

- A return statement with a value should not use parentheses unless they make the return value more obvious in some way.

```
return;
return myDisk.size();
return (size ? size : defaultSize);
```

return Statements

- Try to make the structure of your program match the intent

```
if (booleanExpression) {
 return TRUE;
} else {
 return FALSE;
}
```

- should instead be written as

```
return booleanExpression;
return (condition ? x : y);
```

```
/*
 * %W% %E% Firstname Lastname
 *
 * Copyright (c) 1993-1996 Sun Microsystems, Inc. All Rights Reserved.
 *
 * This software is the confidential and proprietary information of Sun
 * Microsystems, Inc. ("Confidential Information"). You shall not
 * disclose such Confidential Information and shall use it only in
 * accordance with the terms of the license agreement you entered into
 * with Sun.
 *
 * SUN MAKES NO REPRESENTATIONS OR WARRANTIES ABOUT THE SUITABILITY OF
 * THE SOFTWARE, EITHER EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED
 * TO THE IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A
 * PARTICULAR PURPOSE, OR NON-INFRINGEMENT. SUN SHALL NOT BE LIABLE FOR
 * ANY DAMAGES SUFFERED BY LICENSEE AS A RESULT OF USING, MODIFYING OR
 * DISTRIBUTING THIS SOFTWARE OR ITS DERIVATIVES.
 */
package java.blah;
import java.blah.blahdy.BlahBlah;
/**
 * Class description goes here.
*
 * @version 1.10 04 Oct 1996
 * @author Firstname Lastname
 */
```

```
public class Blah extends SomeClass {
 /* A class implementation comment can go here. */
 /** classVar1 documentation comment */
 public static int classVar1;
 /*** classVar2 documentation comment that happens to be
 * more than one line long
 */
 private static Object classVar2;
 /** instanceVar1 documentation comment */
 public Object instanceVar1;
 /** instanceVar2 documentation comment */
 protected int instanceVar2;
 /** instanceVar3 documentation comment */
 private Object[] instanceVar3;
/**
 * ...method Blah documentation comment...
 */
 public Blah() {
 // ...implementation goes here...
 }
 /**
 * ...method doSomething documentation comment...
 */
 public void doSomething() {
 // ...implementation goes here...
 }
 /**
 * ...method doSomethingElse documentation comment...
 * @param someParam description
 */
 public void doSomethingElse(Object someParam) {
 // ...implementation goes here...
 }
}
```

if, if-else, if-else-if-else Statements

```
if (condition) //AVOID! THIS OMITS THE BRACES {}!
statement;
//----------------
if (condition) {
statements;
}
if (condition) {
statements;
} else {
statements;
}
if (condition) {
statements;
} else if (condition) {
statements;
} else if (condition) {
statements;
}
```

for Statements

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

- An empty for statement (one in which all the work is done in the initialization, condition, and update clauses)

```
for (initialization; condition; update);
```

- When using the comma operator in the initialization or update clause of a for statement, avoid the complexity of using more than three variables.
- If needed, use separate statements before the for loop (for the initialization clause) or at the end of the loop (for the update clause).

while Statements

- A while statement should have the following form:

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

- An empty while statement should have the following form:

```
while (condition);
```

do-while Statements

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

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

switch Statements

```
switch (condition) {
        case ABC:
                statements;
                /* falls through */
        case DEF:
                statements;
                 break;
        case XYZ:
                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 **/* falls 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.

try-catch Statements

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

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

Blank Lines

- 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

Blank Spaces

- A parenthesis should be separated by a space

```
while (true) {
...
}
```

- A blank space should not be used between a method name and its opening parenthesis. .
- 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

Best Programming Practices

- Don’t make any instance or class variable public without good reason.
- Often, instance variables don’t need to be explicitly set or gotten—often that happens as a side effect of method calls.
- public instance variables is the case where the class is essentially a data structure, with no behavior.

Referring to Class Variables and Methods

- Avoid using an object to access a class (static) variable or method. Use a class name instead.

```
classMethod(); //OK
AClass.classMethod(); //OK
anObject.classMethod(); //AVOID!
```

Variable Assignments

- Numerical constants (literals) should not be coded directly, except for -1, 0, and 1, which can appear in a for loop as counter values
- Avoid assigning several variables to the same value in a single statement. It is hard to read.

```
fooBar.fChar = barFoo.lchar = 'c'; // AVOID!
```

- Do not use the assignment operator in a place where it can be easily confused with the equality operator

```
if (c++ = d++) { // AVOID! Java disallows
 ...
}
```

- should be written as

```
if ((c++ = d++) != 0) {
 ...
}
```

- Do not use embedded assignments in an attempt to improve run-time performance. This is the job of the compiler, and besides, it rarely actually helps.

```
d = (a = b + c) + r; // AVOID!
```

- should be written as

```
a = b + c;
d = a + r;
```

Parentheses

- A good idea to use parentheses liberally in expressions involving mixed operators to avoid operator precedence problems.
- Even if the operator precedence seems clear to you, it might not be to others—you shouldn't assume that other programmers know precedence as well as you do.

```
if (a == b && c == d) // AVOID!
```

```
if ((a == b) && (c == d)) // RIGHT
```