Java Coding Conventions

File Organization

A file consists of sections that should be separated by blank lines and an optional comment identifying each section. Files longer than 2000 lines are cumbersome and should be avoided

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. Java source files have the following ordering:

- Beginning comments
- Package and Import statements; for example:
  ```
  import java.applet.Applet;
  import java.awt.*;
  import java.net.*; •
  ```
- Class and interface declarations

Beginning 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. For example:

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

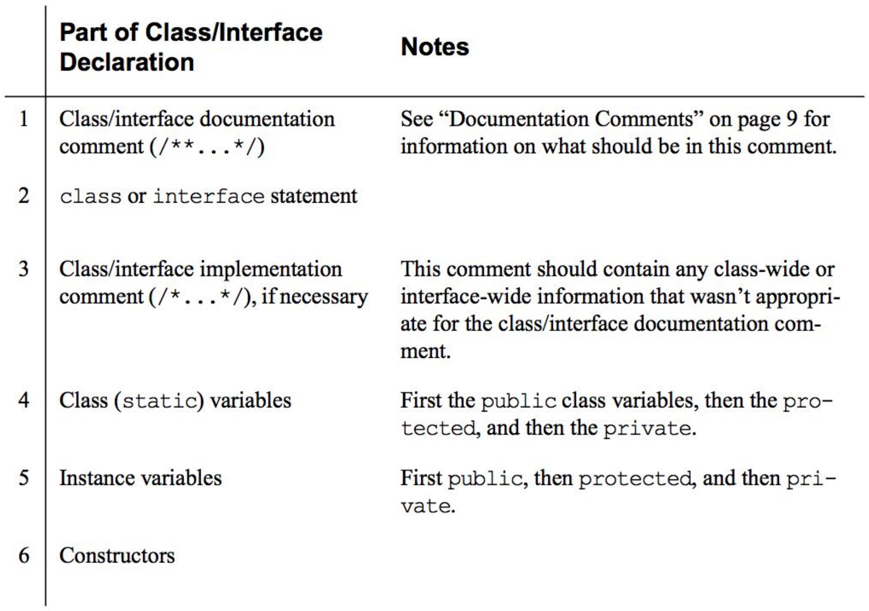
Package and Import Statements

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

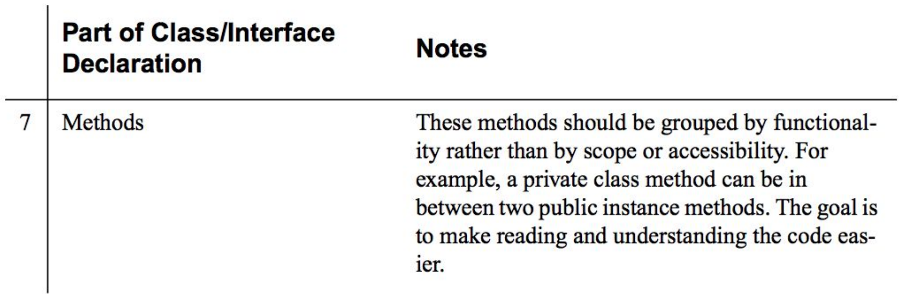
```
package java.awt;
 import java.awt.peer.CanvasPeer;
```

Class and Interface Declarations

The following table describes the parts of a class or interface declaration, in the order that they should appear.



	Part of Class/Interface Declaration	Notes
1	Class/interface documentation comment (`/**...*/`)	See "Documentation Comments" on page 9 for information on what should be in this comment.
2	`class` or `interface` statement	
3	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.
4	Class (`static`) variables	First the `public` class variables, then the `protected`, and then the `private`.
5	Instance variables	First `public`, then `protected`, and then `private`.
6	Constructors	



	Part of Class/Interface Declaration	Notes
7	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.

Indentation

1 Tab 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 4 spaces (not 8).

Line Length

Avoid lines longer than 80 characters, since they're not handled well by many terminals and tools.

Comments

Java programs can have two kinds of comments: implementation comments and documentation comments. Implementation comments are those found in C++, which are delimited by /*...*/, and //. 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 are mean 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. Comments should be used to give overviews of code and provide additional information that is not readily available in the code itself. Comments should contain only information that is relevant to reading and understanding the program. For example, information about how the corresponding package is built or in what directory it resides should not be included as a comment. Discussion of nontrivial or non-obvious design decisions is appropriate, but avoid duplicating information that is present in (and clear from) the code. It is too easy for redundant comments to get out of date. In general, avoid any comments that are likely to get out of date as the code evolves. Note: The frequency of comments sometimes reflects poor quality of code. When you feel compelled to add a comment, consider rewriting the code to make it clearer. Comments should not be enclosed in large boxes drawn with asterisks or other characters. Comments should never include special characters such as form-feed and backspace.

Block comments

These 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.
*/
```

Block comments can start with /*-, which is recognized by indent as the beginning of a block comment that should not reformatted. Example:

```
/*
 * Here is a block comment with some very special
 * formatting that I want indent(1) to ignore.
 *
 * one
 * two
 * three
 */
```

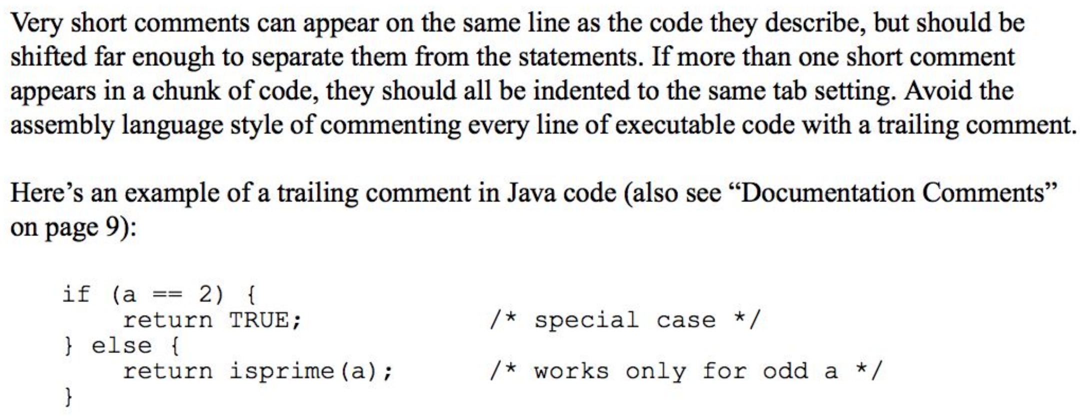
Single-Line Comments

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 (see section 5.1.1). A single-line comment should be preceded by a blank line. Here’s an example of a single-line comment in Java code :

```
 if (condition) {
```

/* Handle the condition. */
…
}

Trailing Comments



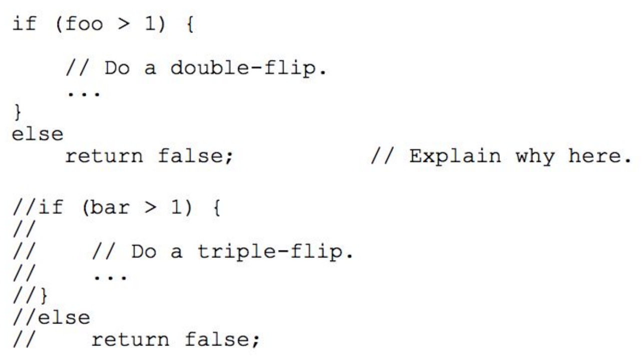
Very short comments can appear on the same line as the code they describe, but should be shifted far enough to separate them from the statements. If more than one short comment appears in a chunk of code, they should all be indented to the same tab setting. Avoid the assembly language style of commenting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

Here's an example of a trailing comment in Java code (also see "Documentation Comments" on page 9):

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

End-Of-Line

Comments 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. Examples of all three styles follow:



```
if (foo > 1) {

    // Do a double-flip.
    ...
}
else
    return false;           // Explain why here.

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

Documentation Comments

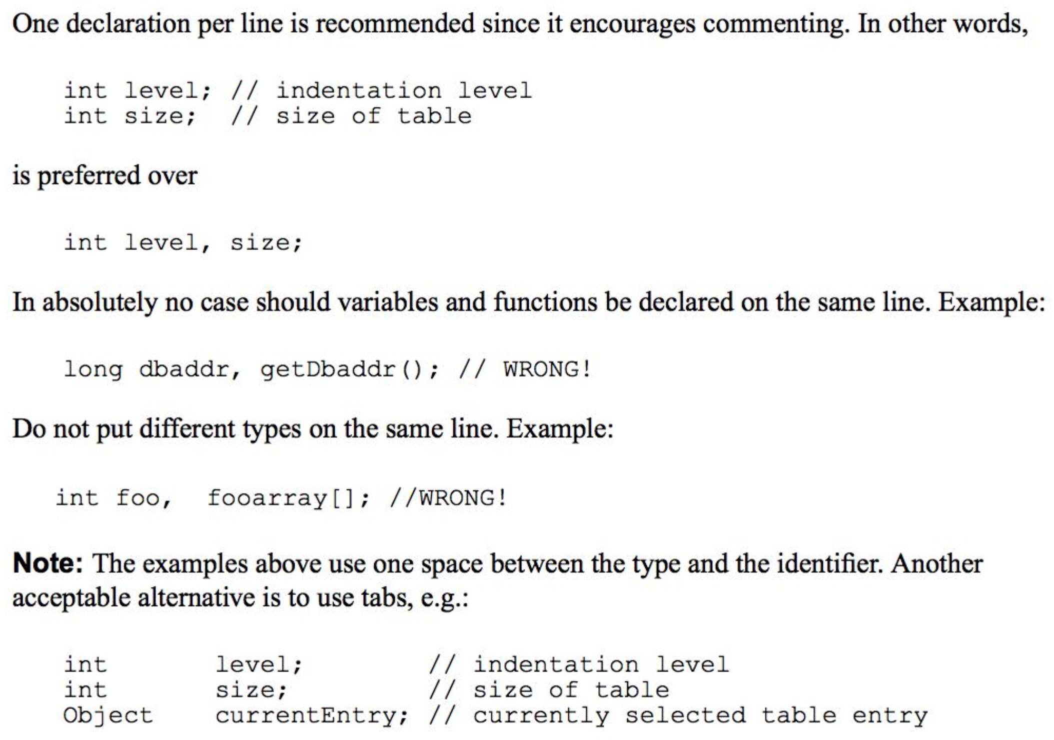
Doc comments describe Java classes, interfaces, constructors, methods, and fields. Each doc comment is set inside the comment delimiters /**...*/, with one comment per API. This comment should appear just before the declaration:

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

class Example { …
Notice that classes and interfaces are not indented, while their members are. The first line of doc comment (/**) for classes and interfaces is not indented; subsequent doc comment lines each have 1 space of indentation (to vertically align the asterisks). Members, including constructors, have 4 spaces for the first doc comment line and 5 spaces thereafter. If you need to give information about a class, interface, variable, or method that isn't appropriate for documentation, use an implementation block comment or single-line comment immediately after the declaration.

Declarations

Number Per Line



One declaration per line is recommended since it encourages commenting. In other words,

```
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. Example:

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

Do not put different types on the same line. Example:

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

Note: The examples above use one space between the type and the identifier. Another acceptable alternative is to use tabs, e.g.:

```
int       level;         // indentation level
int       size;          // size of table
Object    currentEntry;  // currently selected table entry
```

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. For example, do not declare the same variable name in an inner block:

```
int count;
...
func() {
    if (condition) {
        int count;      // AVOID!
        ...
    }
    ...
}
```

Class and Interface Declarations

When coding Java classes and interfaces, the following formatting rules should be followed:

- 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

Statements

Simple Statements

Each line should contain at most one statement. Example:

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

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

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

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 a `if-else` or `for` statement. This makes it easier to add statements without accidentally introducing bugs due to forgetting to add braces.

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();

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

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 if (condition) {
    statements;
}
```

for Statements

A `for` statement should have the following form:

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

An empty `for` statement (one in which all the work is done in the initialization, condition, and update clauses) should have the following form:

```
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);
```

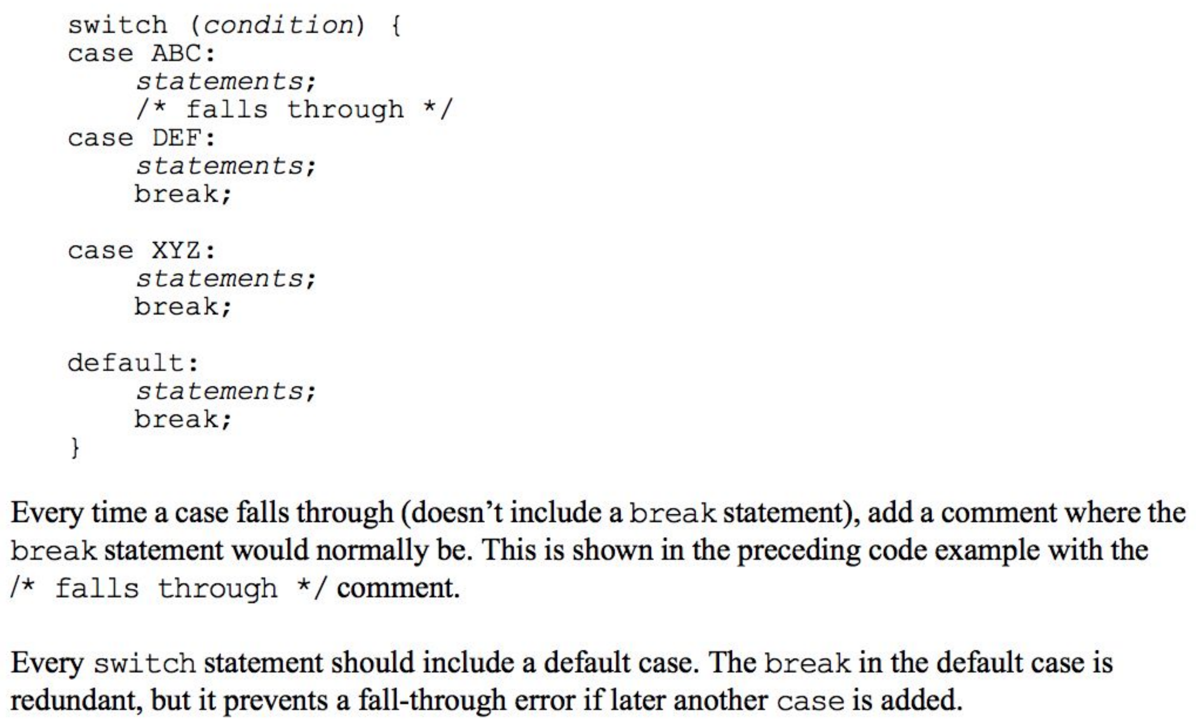
do-while Statements

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

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

switch Statements

A `switch` statement should have the following form:



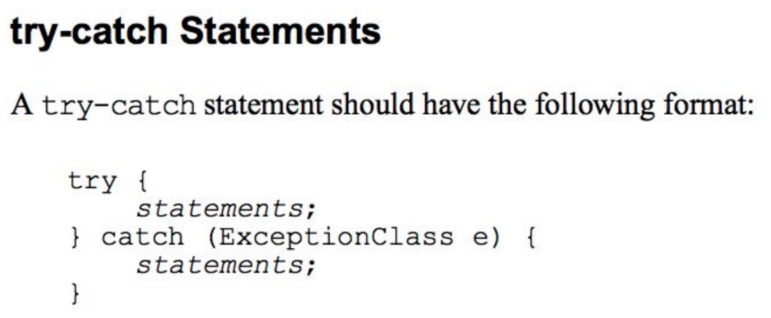
```
switch (condition) {
case ABC:
    statements;
    /* falls through */
case DEF:
    statements;
    break;

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.



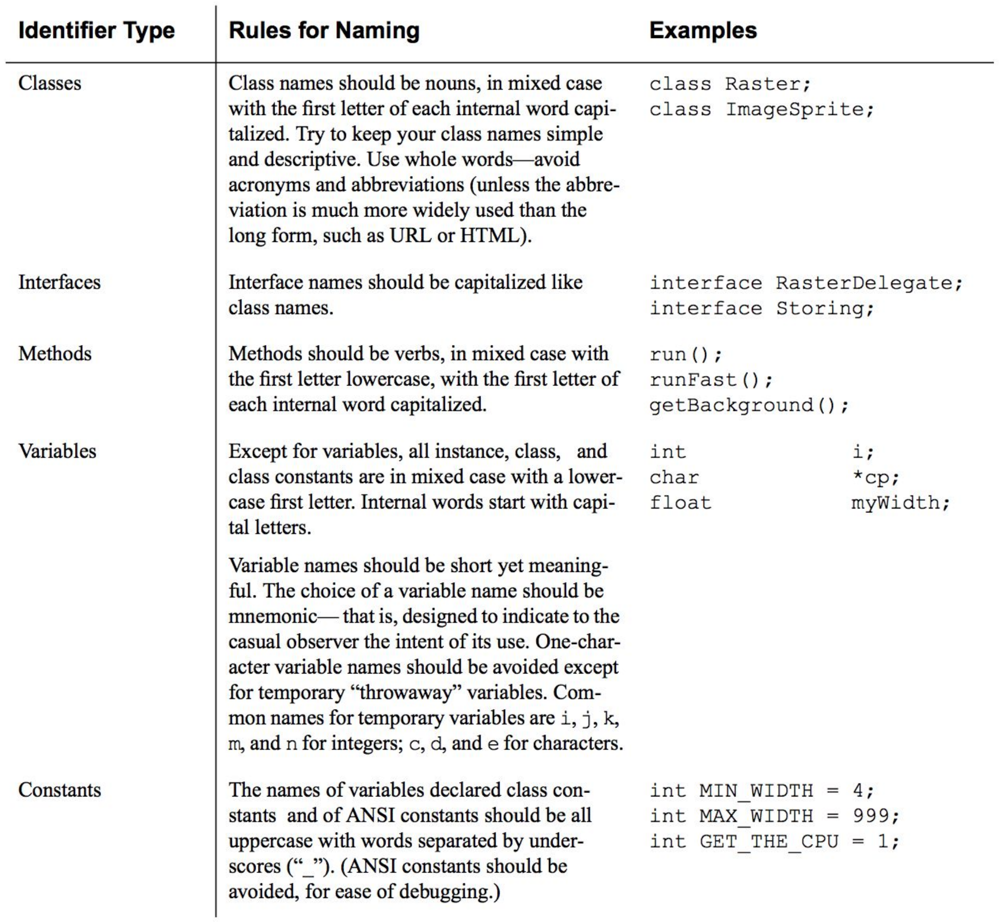
try-catch Statements

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

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

Naming Conventions:

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



Identifier Type	Rules for Naming	Examples
Classes	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;`
Interfaces	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();`
Variables	Except for variables, all instance, class, and class constants are in mixed case with a lowercase first letter. Internal words start with capital letters. Variable names should be short yet meaningful. The choice of a variable name should be mnemonic— that is, 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.	`int i;` `char *cp;` `float myWidth;`
Constants	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;`

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
- Between logical sections inside a method to improve readability

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 (true) {...

```
}
```
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 (d++ = s++) {
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);
```

Programming Practices

Providing Access to Instance and Class Variables

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.

One example of appropriate public instance variables is the case where the class is essentially a data structure, with no behavior. In other words, if you would have used a `struct` instead of a class (if Java supported `struct`), then it's appropriate to make the class's instance variables public.

Referring to Class Variables and Methods

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

```
classMethod();              //OK
AClass.classMethod();       //OK

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

Constants

Numerical constants (literals) should not be coded directly, except for -1, 0, and 1, which can appear in a `for` loop as counter values.

Variable Assignments

Avoid assigning several variables to the same value in a single statement. It is hard to read. Example:

```
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. Example:

```
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. Example:

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

should be written as

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

Miscellaneous Practices

Parentheses

It is generally 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
```

Returning Values

Try to make the structure of your program match the intent. Example:

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

should instead be written as

```
return booleanExpression;
```

Similarly,

```
if (condition) {
    return x;
}
return y;
```

should be written as

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

Expressions before '?' in the Conditional Operator

If an expression containing a binary operator appears before the `?` in the ternary `?:` operator, it should be parenthesized. Example:

```
(x >= 0) ? x : -x
```

Special Comments

Use `XXX` in a comment to flag something that is bogus but works. Use `FIXME` to flag something that is bogus and broken.