

Notes on Programming Style

One of the major goals of this course is that you learn to write programs that are not only correct, but that are readable and understandable. These guidelines should help you towards that goal. In this course we ask that you follow the rules given here. We hope they will give you a good basis for developing a style of your own.

1 Why?

So why all the fuss about programming style, comments, and variable names? Is it just one of those odd things that instructors require in courses or does it really matter?

A computer program has two distinct roles. The most obvious is to instruct the computer to perform some useful task — format and print a paper, simulate a factory, translate Java code into assembly language. Less obvious, but far more important, is that it is a document that must be readable and understandable. New computers, operating systems, and software upgrades appear constantly; the needs of users change over time; new laws are passed; companies merge and split up. Programs must be changed to adapt.

A working program is the result many design decisions. These decisions — why this variable was introduced, what that method does — are not reflected in the final code, which consists of very low-level, detailed declarations and statements. But the higher-level design *must* be understood to successfully alter a program. If these decisions are not recorded in comments, they must be reconstructed. This technological archeology is painful, aggravating, annoying, and error-prone — and all too common.

“But this is only a quick hack. I’ll never use it again. I don’t need to waste my time on comments.” Wrong. Every useful program lives forever, often in spite of the author’s original intentions. A few months from now you won’t remember the details that went into writing it. If you have to do something to the program and it was poorly written or lacks good comments, you will have a miserable time.

“It’s a waste of time to put the comments in now. I’m going to change the program when I debug it and I don’t want to have to change the comments too. I’ll add the comments once the program is working right.” Wrong. Programmers, even those with the best of intentions, never go back and fill in the comments. There’s always more code to write, more interesting projects to do. Comments must be included from the start. It takes very little extra effort to add them while you’re writing the code, and the resulting comments are better than anything added after the fact. The information that belongs in a comment is freshest when a variable or method is first created. It is much harder to try to reconstruct it later.

Writing comments as you write the code will help you clarify your ideas and write better, correct code sooner. If you can write down clearly what your program is doing, you are more likely to have a good understanding of the problem and your code is more likely to be correct. Time spent on careful thinking and writing is more than repaid in time saved during testing and debugging.

Good comments can also help you find or avoid errors (bugs). A very common source of errors is inconsistent use of variables and method parameters. If the precise definition of a variable is given in a comment, you can verify that each reference to the variable is consistent with the definition. If a precise definition isn’t available, you are likely to use the variable in one way today and in a slightly different way tomorrow, which creates subtle, hard to find errors.

2 General Guidelines

The most important rule is that every method, data structure, and significant variable must be given a complete definition in a comment that accompanies its declaration. This comment should contain *everything* needed to understand the item and how to use it *and no more* — don't include unnecessary implementation details that should be private. Comments should be complete. But they may refer to a published handout, article, book, or manual if the full definition is available elsewhere.

It should *never* be necessary to look at code that uses or implements a method, variable, or data structure to understand how to use it (as opposed to understanding how it's implemented). If you find that you have to read client code to understand a method or variable, then the comments are inadequate.

The complete definition should appear in only one place. Duplicating information in more than one place invites trouble when the program is changed. It is all too easy to fix one comment and forget to fix a related one elsewhere. Exception: the implementation of a method should include a complete specification comment even if that specification also appears in a comment at the beginning of the class definition.

Comments are needed to break up long sequences of statements. Indenting and blank lines and other whitespace should be used to clarify the structure of the code and avoid clutter.

If you are modifying code originally written by someone else, match their style. It will help the next person who has to read the program if a consistent style is used throughout.

The comments must agree with the program. False comments are worse than none at all.

3 Class Headings

Every file in your code should begin with a short comment giving the name of the file, the authors, and a very high-level description of its contents.

```
// Scanner.java -- Lexical analyzer for C--
// Al Gaulle, 6/8/60
```

Other useful information in the heading is a brief summary of the public methods and variables (i.e., the class's interface) and 1-line descriptions of major changes to the class. For course assignments, the file heading should include the course name, assignment number, and any other information requested in the assignment.

4 Names

Naming is one of the most important parts of programming. Good names make a program more readable and can reduce the amount of other documentation needed.

Methods and nontrivial variables must be given meaningful names. An appropriate name for a variable or method that returns a value is a noun or noun phrase describing the contents of the variable or value computed by the method (e.g., `length`, `total_sales`, `currentInventory`). A good name for a method that is only executed for its effect (i.e., has a result type of `void`) is a verb or verb phrase describing the action performed (`print_report`, `ringBell`).

Names should be neither too long nor too short. Names that are significant to the problem being solved should have descriptive names as should names defined in libraries for use by others. Avoid cryptic abbreviations. Use `sales_tax` or `salesTax`, not `stax`.

For variables and parameters that are only used in a small method or region of the program, a short name is often better than a long one.

```

// yield larger value of x and y
public int max (int x, int y) {
    return x > y ? x : y;
}

// print a line of n *'s
public void printStars(int n) {
    for (int k=1; k<=n; k++)
        System.out.print("*");
    System.out.println();
}

```

A name like `theLoopCounter` instead of `k`, or `firstNumber` and `secondNumber` for `x` and `y` would only create clutter. But avoid large collections of cryptic names. A method with parameters named `xx`, `xx1`, `ff1`, `ff`, and `fff` will be hard to understand.

A variable used as a “flag” should not be named `flag` but should be named for what the flag represents like `noMorePizza`. Avoid generic names like `count` and `value`. Instead, describe the items being counted or the value stored in the variable.

Use consistent naming conventions. The same thing should usually have the same name when it appears in more than one place. Avoid vague, misleading, silly, or obscene names (it’s already been done). The amount of pizza in the fridge should not be named `stuff` or `cat` or `fred` (even if it’s Fred’s pizza).

In Java, compound names can be written as either `compoundName` or `compound_name`. Traditional typography would suggest using the underscore, because upper-case letters were not designed to appear immediately to the right of a lower-case letter. But embedded upper-case letters are widely used in computing and have rubbed off on the mass culture, resulting in the biZarRE CAPitAlIZaTiOn found in much advertising and display text these days. Most Java books favor embedded upper-case letters. Take your choice (of the underscore vs embedded caps; not the wEIrD stuff).

Capitalization is significant in Java: `thisid`, `thisId`, and `ThisId` are three different identifiers. A useful convention is to begin all identifier and method names with lower-case letters and all class names with upper-case. Used consistently, this makes code easier to read and reduces clashes between type and variable names.

5 Data Definitions

Every significant variable and data structure needs a precise and complete definition. This should provide any information needed to understand the variable in addition to its name and type. The most useful information is often *invariant properties* of the data: facts that are always true except, perhaps, momentarily when several related variables are being update. For arithmetic variables this might be a formula showing how the variables are related (`// 1 <= currentItem <= maxItems`). For a variable holding a logical value, it is often easiest to define it with a phrase that gives its meaning when true.

```
boolean done;           // = "user has selected Quit from file menu"
```

Definitions must be precise. Comments like “flag for loop” or “index into array b” say nothing. Instead, describe the condition the flag represents or, if `i` is used as a subscript for array `b`, explain what `b[i]` is. A definition like “error code” is not complete. If particular values of a variable have specific meanings, those values and their significance must be described.

Variables used only as loop indices or subscripts do not need a comment if none would be helpful.

Related variables should be declared and described together. For example, the definition of a table should describe not only the array that holds the data but also the integer variable containing the number of items currently in the table.

```

static final int maxTemps = 150; // maximum # of temperature readings

double [ ] temps = new double[maxTemps];
                        // Temperature values are stored
int      nTemps;      // in temps[0..nTemps-1].

```

These comments should appear to the right of the variable declarations. Use tabs to line up identifier names and comments. Don't run everything together; it is much harder to read.

```

double[ ] temps=new double[maxTemps]; // Temperature values are
int nTemps; // stored in temps[0..nTemps-1].

```

Related variables should normally be packaged as an object. Comments describing the fields of an object belong with the class definition; comments beside variables of that class should describe the contents of that variable, not the object fields.

```

Class Temps {                // Table of temperature readings:
    double [ ] temps;        // Temperature values are stored
    int      nTemps;        // in temps[0..nTemps-1].

    // construct Temps object with a table having n entries
    public Temps(int n) {
        temps = new double[n];
        nTemps = 0;
    }
};

Temps ITH = new Temps(20);    // temperatures in Ithaca
Temps HNL = new Temps(50);    // temperatures in Honolulu

```

6 Method Specifications

Every method should be preceded by a comment giving its specification. This specification and the prototype of the method, which gives the number and types of the parameters and type of the result, should provide *all* of the information needed to use the method and no more. It should describe *what* the method does, not *how* it does it. One should *never* have to look at the body of a method to understand how to use it. The specification comment is the place where the parameters of the method are described. All of this can usually be worked into a sentence or two.

```

// Yield the temperature found most frequently in table t
public double find_common (Temps t)

```

It is, unfortunately, more typical to find a comment like this, if any comment is provided at all.

```

// Find most frequent temperature
public double find_common (Temps t)

```

What is the purpose of parameter `t`? Does the method change it? If so, how? What value is returned by method `find_common`? The comment doesn't say.

The heading must be complete. But be concise. Don't write an essay if a short sentence will do.

Use the active voice.¹ Omit needless words.² Don't write "Method to crash the car..." or "Method crumple crashes the car..." or even "Crashes the car...". Just say "Crash the car."

¹W. Strunk, Jr., and E. B. White, *The Elements of Style*, 3rd ed., rule 14, p. 18. This book is required reading for all writers, including programmers.

²Strunk and White, rule 17, p. 23.

For a value-returning method, it is often easiest to simply describe the value returned.

```
// = distance between points (x1,y1) and (x2,y2)
public double dist (int x1, int y1, int x2, int y2);
```

7 Statement Comments

Comments should be included in long sequences of statements to describe logical units of processing. A “statement comment” should be a higher-level description of the operation implemented by the group of statements beneath it.

```
// Ensure x >= y, exchanging the values of x and y if needed.
    if (x < y ) {
        tmp = x;
        x = y;
        y = tmp;
    }
```

The comment should explain *what* the group of statements does, not *how* it does it. Comment groups of statements, not individual statements whose meaning is clear. Put a blank line before such comments to visually separate these paragraph-like chunks of code.

These comments can be a great help to someone trying to understand the program since they document its high-level (“top-down”) structure, which is not otherwise visible. They also help a reader scan the program quickly to find the section of current interest, much like the section and paragraph headings in a book or article.

Statement comments must be complete. The comment

```
// Test for valid input
```

is not adequate. What happens if the input is valid? What if it isn’t? The comment should include this information.

```
// Make a rude noise and terminate execution if the input is not valid.
```

Obscure or unusual code should be avoided but when necessary a comment should be used to clarify.

```
// round cents to nearest dollar
    cents = 100 * ((cents+50) / 100);
```

If a complex algorithm or data structure is being implemented, a block of comments describing it, or a reference to other sources of information (books, articles, etc.), should be included above the data structure or method definitions.

Do not comment that which is already clear. Don’t write

```
// print the gross sales amount
    System.out.println("Gross sales = " + gross_sales);
```

or

```
// increment k
    k++;
```

Assume that the reader knows Java at least as well as you do. Comments should not be used to explain how the programming language works.

Statement comments should be placed above the code they document, not out to the side. Such marginal comments usually wind up paraphrasing the code without adding useful information.

```

k = a[i];                // look at the next number
if (k < 0)                // check if it's negative
    System.out.println("complain"); // print error message if it is

```

Textbook authors sometimes do this to explain example code. In real programs it is useless clutter. Don't do it.

Exception: In long files, comments in the right margin can serve as useful “tab” markers to help the reader skim through the code. An example is a switch statement that extends over several pages.

```

switch (token.class) {
    case ident: ...           // identifier
                break;
    case int:   ...           // integer
                break;
    case lparen: ...         // left parenthesis
                break;
    default:   ...           // unknown
}

```

Avoid redundant comments. Say things once in the proper place rather than repeatedly throughout the program. It is very possible to obscure a program by over-commenting. More is not necessarily better. Your purpose in writing is to guide your readers and anticipate questions they might have. Include enough to do this and no more.

8 Coding Conventions

This section contains several low-level details that need to be attended to. These rules are not necessarily better than any others but they will lead to readable code, so we ask you to use them in your programs.

8.1 Indenting

Programs should be indented to make them easier to understand. Indenting of variable declarations was covered earlier. The bodies of methods, loops, and conditional statements should be indented to make the logical structure clear.

If one were to pick a single piece of syntactic trivia that is responsible for more pointless heated debate among Java, C, and C++ programmers, it is probably where to put the left curly brace at the beginning of a compound statement. One possibility is at the end of the previous line.

```

if (x < y) {
    x = y;
    y = 0;
} else {
    x = 0;
    y = y/2;
}

```

Another is to put it on a line by itself.

```

if (x < y)
{
    x = y;
    y = 0;
}
else

```

```

{
    x = 0;
    y = y/2;
}

```

Which is best is mostly a matter of religious preference. The former has the advantage of conserving vertical space, which helps fit more code onto a single screen or page. The latter has a more pleasing symmetry. Some style guides suggest putting the curly brace that begins a method body on a line by itself and putting other left braces at the end of a line.

Pick one style and use it consistently. If you're working on code written by someone else, match their style.

8.2 Symbolic Constants

Important constants should be given symbolic names and these names should be used throughout the code instead of the numeric value. This is particularly true for physical constants and parameters related to the problem being solved.

```

static final double G = 6.67e-11;    // Gravitational constant

static final int max_grades = 200;   // Maximum # grades in input

```

Using a symbolic name reduces the possibility of typographical errors. It also makes it much easier to change values when needed. If the constant `max_grades` is used throughout the program, it is easy to adjust the maximum by changing the initial value in the declaration. But if 200 is used directly, it cannot be changed without scanning the code for every occurrence of 200, deciding if that occurrence refers to the maximum number of grades, and changing it if it does. It is very easy to miss a copy, change one that should be left alone, or miss a related number.

Java programmers tend to capitalize the names of symbolic constants: `MAX_GRADES`. If you are working on code that follows this convention, do the same.

8.3 Block Comments

Programmers have strong preferences about how to write comment blocks, like the heading comment that should appear at the beginning of each file. Possibilities include:

```

// Stuff.java -- class Stuff and other useful things
// hp, 9/96

//-----
// Stuff.java -- class Stuff and other useful things
// hp, 9/96
//-----

/*****
/* Stuff.java -- class Stuff and other useful things */
/* hp, 9/96 */
*****/

/*
 * Stuff.java -- class Stuff and other useful things
 * hp, 9/96
 */

```

Pick a style that looks good to you and use it.

9 Acknowledgments

The ideas in this missive originated in the Structured Programming movement of the 1970's and are every bit as applicable today. Many specific examples are taken from old CS100 and CS211 handouts, originating in the work of David Gries and Richard Conway, and modified by many authors over the years. Other good ideas and explanations came from the *Tcl/TK Engineering Manual* by John Ousterhout of Sun Microsystems, which is an industrial-strength style guide for a large collection of C code.