> *[D]ocumentation is your secret weapon, the unsung hero of your startup, ... which keeps things going behind the product development.*
> — Vadim Kravcenko
> `https://vadimkravcenko.com/shorts/proper-documentation/`

Introduction

A computer program is a detailed description of *how* to solve a problem, detailed enough that a computer **compiler** can turn it into a sequence of instructions that the computer can execute on any instance of the problem, solving it.

The *programmer*, to write the *program*: understands the **problem**, understands how to **solve** the problem, and understands how to **describe** how to solve the problem. When the problem specification changes (as they always do), the future programmer must rebuild this knowledge for the *existing* code and determine how to map that **old** solution onto a **new** solution.

Good program *design* makes a program easier to *understand*; good coding *standards* make a program easier to *read* so that the design can be seen.

This document is written as a collection of coding standard *rules*. Each subsection begins with a statement of a rule, an example of applying the rule, and then a discussion of the motivation of the rule.[1] The sections collect related rules into groups.

Readability

You write a computer program for the first time *once*. A program is **read** more than ten-times more often (by you during debugging, during code review, during modification, *etc.*) than it is **written**. It is therefore *cost-effective* to make the code as physically easy-to-read. For yourself and any future programmer (who may well be you, too).

Formatting

Reading is a mechanical process first. You need to make sure your code guides the reader's eye across the page as efficiently as possible.

Lines

Rule Lines should never be longer than 80 characters.
Example Compare the following two listings of the same code segment:

```
while (charCount == 0) {
    for (int i = offset; i < offset + charCount; i++) {
        if (!this.inMultiLineComment && !this.inSingleLineComment && !this.ignoreInput) {
            if (destinationBuffer[i] == '#' && i < offset + charCount - 1 && destinationBuffer[i+1] == '|') {
                this.inMultiLineComment = true;
            } else if(destinationBuffer[i] == ';') {
                this.inSingleLineComment = true;
            } else {
                destinationBuffer[last++] = destinationBuffer[i];
            }
        }
    }
}
```

```
while (charCount == 0) {
    for (int i = offset; i < offset + charCount; i++) {
        if (!this.inMultiLineComment &&
            !this.inSingleLineComment &&
            !this.ignoreInput) {
            if (destinationBuffer[i] == '#' &&
                i < offset + charCount - 1 &&
                destinationBuffer[i+1] == '|') {
                this.inMultiLineComment = true;
            } else if(destinationBuffer[i] == ';') {
                this.inSingleLineComment = true;
```

[1] Dr. Ladd first encountered this presentation order in Koenig and Moo's *Accelerated C++*.

```
            } else {
                destinationBuffer[last++] = destinationBuffer[i];
            }
        }
    }
}
```

Motivation Reading involves scanning lines of text (code). Lines that are too wide for reading in about two glances slow down processing. Also, as in the example above, printing out code that is too wide can go beyond the width of a page and lose information.
Exceptions Import lines go on a single line.

Rule Never break `import` or `package` lines.
Motivation Long package lines do not often go beyond the 80 character limit but if they do, there is no *good* way to determine where they should be broken. There are also few of them and they appear at the top of the file so they do not break reading flow.

Rule Indentation **must** be consistent in source files. Always use the *space* character for indentation, never the *tab* character. [2]
Example It the following code, what is the body of the loop?

```
void readFile(String fname) {
try {
       Scanner fin = new Scanner(new File(fname));
 int wordCounter = 0;
       while (fin.hasNext()) {
     String word = fin.next();
         wordCounter += 1;
         System.out.println(”[%d] %s”, wordCounter, word));
    }
     } catch (IOException e) {
   e.printStackTrace();
}
}
```

Yet, the same code can be made easier to read with consistent 2-space or 4-space indentation:

```
void readFile(String fname) {
  try {
    Scanner fin = new Scanner(new File(fname));
    int wordCounter = 0;
    while (fin.hasNext()) {
      String word = fin.next();
      wordCounter += 1;
      System.out.println(String.format(”[%d] %s”,
                                       wordCounter, word));
    }
  } catch (IOException e) {
    e.printStackTrace();
  }
}
```

```
void readFile(String fname) {
    try {
        Scanner fin = new Scanner(new File(fname));
        int wordCounter = 0;
        while (fin.hasNext()) {
            String word = fin.next();
            wordCounter += 1;
            System.out.println(String.format(”[%d] %s”,
                                             wordCounter, word));
        }
    } catch (IOException e) {
        e.printStackTrace();
    }
}
```

Motivation Indentation in programming languages shows *structure*. In `Python`, whitespace **is** structure. Consistent indentation guides the reader across details they want to just scan and helps them drill down on the details deemed important.

Rule Use *enough* blank lines to structure your code but do not use *too many* blank lines.
Motivation No example here, just some rules of thumb.
Set the `package` line off from the header comment with a single blank line.
Set the `import` lines off from the `class` header comment with a single blank line.
Consistently put zero (0) or one (1) blank line between data members in classes. If use no whitespace between lines, make sure to offset the data with a blank line from the rest of the class.

[2] Except in `Makefile`s and other formats that give meaning to the *tab*.

Consistently put one (1) or two (2) blank line(s) between methods in a class.
It is often a good idea to set a loop (top and bottom) or conditional (top and/or bottom) from the rest of the code by a single blank line. This is important in long methods (and might be better addressed by breaking long methods into shorter, simpler, single-purpose methods).

Rule Opening curly braces, `{`, should *never* be alone on a line; closing curly braces, `}`, are alone on a line more often than not.

Example Bad code!

```
public class Cubic
  extends Square
{
  public static void main(String[] args)
  {
    if (args.length != 1)
    {
      System.err.println(”Call w/ exactly one file name.”);
      System.exit(1);
    }
    else
    {
      // ... code
    }
  }
}
```

Good code!

```
public class Cubic
  extends Square {
  public static void main(String[] args) {
    if (args.length != 1) {
      System.err.println(”Call w/ exactly one file name.”);
      System.exit(1);
    } else {
      // ... code
    }
  }
}
```

Motivation This is a topic open to heated debate in the realm of software engineering. The concluding advice in most arguments for a given style is that a smart programmer learns to use the formatting rules of the shop where they program.
You program in the CS Department and this is the given style for curly braces.

Ordering: File

Source files should follow consistent convention so that the programmer reading the code knows where to look for necessary information.

Rule Sections in a Java source file should be ordered as follows. Skip any empty section.

```
<Package Declaration>

<Header Comment>

<Import Statements>

<Class Declaration>
```

Example The following is a minimized example without an explicit package nor any import statements.

```
/**
 * Hello, World in a minimized class.
 *
 * Hello, World is a 5 decade old example. This in default package
 * @author Brian C. Ladd
 */

public class HelloWorld {
  public void main(String args[]) {
    System.out.println(”Hello, World!”);
  }
}
```

Motivation The relative ordering of everything but the comment is dictated by the Java programming language. The package comes first because many editors look for it there. The header comment is probably the first think a programmer wants to see upon opening your file.

Rule A `.java` file must contain a *single* `class` definition.
Motivation Java requires that a `public` class be declared in a file with the same name as the class. It is *possible* to declare (and use) non-`public` classes outside that class in the same file.
Never do this.
It makes finding the hidden class definition impossible for a future programmer.
Note: you may declare any *inner* classes inside the single, top-level, `public` class.

Ordering: Class

One must navigate within class definitions. A structure serves as a skeleton on which you hang your code.

Rule Fields/methods in a class should be ordered as follows. Skip any empty section. Within any given section, order alphabetically by the names of the fields or methods. With overloaded methods (including constructors) order by number of parameters.

```
public class <Class Name>
  extends <Parent>
  implements <InterfaceA>, <InterfaceB>, ... {

  <Private Field Declarations>

  <Constructors>

  <Public Methods>

  <Private Methods>
}
```

Motivation Java is very permissive on ordering (as with blank lines and other whitespace). Java source code does not support the idea of a table of contents so this ordering convention makes it easier to scan for the part of the class that one suspects is of interest.

Comments

The first (above the source code) level of documentation is the comments in a source file. Header comments are required and must use JavaDoc formatting.

In-line comments clarify complex code — if you are writing lots of these comments, consider if you can write less complex code and guide the reader to your thinking/intent.

Rule All header comments must use `JavaDoc` formatting and styling.
Example The `/**` beginning of the comment is intentional; it is how the `JavaDoc` program can tell which comments to process (those with the double star) and which to skip (those with a single star in the beginning).

```
/**
 * @author Jimmy A. Student
 * @email studeja199@potsdam.edu
 *
 * @param n the non-negative integer to calculate the factorial for
 * @note n is NOT checked as being non-negative; caveat emptor
 * @return n! or n factorial
 */
```

Motivation The fields, marked by `@`, such as `@author`, mark the names of fields that could be extracted by the `JavaDoc` program when creating HTML documentation.
The `@param` and `@return` are important in that they document how to pass information into and read information back from a method.
Exceptions

Rule **Every** source code file must begin with a *file header comment*
Example Imagine that the file `Gargoyle.java` is part of the project solution. That file must begin with the following

```
/**
 * Gargoyle draws a random ASCII art monster on standard output.
 *
```

```
 * Gargoyle has all static methods (and no constructor) including
 * main. It is run with a single integer on the command-line that
 * is used to randomize the monster that is generated.
 *
 * @author Jimmy A. Student
 * @email studeja199@potsdam.edu
 * @course CIS 203 Computer Science II
 * @assignment 4
 * @due 04/25/2018
 */
```

Motivation A file header comment has three parts: summary (the first line), detailed description, and developer identification block.
The summary and detailed description document the *intent* of the class defined in this `.java` file.
More than just **repeat** the name of the class, the summary indentifies, in eighty characters or less, **why** this class exists. The reader gets early guidance if this class contains the functionality they need to see.
In comparison to the summary, the *detailed* description expands on **what** the class does, **how** a programmer would use it, and lists the **collaborator** classes in the solution.
As mentioned earlier, the **id block** uses keywords starting with the ampersand (`@`) because these are treated specially by the `javadoc`[3] program. This id block is *required* in all CS classes. Files without an id block will *not* be graded and given a 0 (zero).

Rule Every method **must** have a header comment with the parameters and return value documented in `JavaDoc` format.
Example A very complete method header comment. Note the `@throws` keyword in particular.

```
/**
 *  Turns gargoyle drawing clockwise.
 *
 *  Rotates the monster being drawn by the number of
 *  degrees. Cannot turn 360.0 degrees or more nor
 *  less than 0.0 (one rotation).
 *
 * @pre 0.0 <= facing < 360.0
 * @post 0.0 <= facing < 360.0
 * @param degreesOfTurn how far to turn the monster; on the range
 * [0.0 - 360.0)
 * @return the new facing, normalized to [0.0 - 360)
 * @throws IllegalArgumentException if the argument is outside
 * specified range
 */
pubilc static double turnClockwise(double degreesOfTurn)
  throws IllegalArgumentException {...}
```

Motivation Like the file header comment, this comment begins with a summary and a detailed description. The *summay*, at a higher-level of abstraction, documents the *intent* of this method, **why** it exists: "What does this method do?"[4] The description documents parameters and calling conventions: "How is this method used?"
Keywords appear at the *end* of the header comment. *All* of the keywords are described below in the order they appear in a header comment, including keywords not used in the example. Skip keywords that do not apply to the method.

`@pre` The required *preconditions*. What must be true for the method to perform correctly. In non-static methods "The object has been constructed" is an implicit precondition that need not be written.
`@post` The promised *postcondition*. What is always true when the method finishes?
`@param` A *parameter* passed in to the method. Name it, then describe what it means and include limitations on expected range of values passed in. There is one `@param` for each parameter passed in to the method.
`@return` Describe the value the method *returns*. Omit when the method returns `void`.
`@throws` If the method has a `throws` clause, name the exception thrown and describe the conditions under which it is thrown. There is one `@throws` keyword for each different type of exception thrown.

---

[3] `javadoc` is a *compiler* like `javac` but instead of handling the Java code and creating a `.class` file, `javadoc` handles the special comments beginning with the `/**` and produces linked Web pages that document the code.

[4] It is often repeated in introductory programming texts that if you cannot come up with a short summary (or a good name) for a method, it is an indication that you are trying to do too much. Compare it to trying to write the topic sentence for a paragraph; if it is hard to do, the paragraph is probably trying to do too much at once and should be broken up.

`@note` A footnote to the header comment. Can be used to describe *implementation* details, note sources that influenced the code, or to explain the design rationale embodied in this method.

Naming

Naming in programs must be "self-documenting", meaning that the identifier is descriptive of the purpose served by the named entity. In general, the names in programs **must** be meaningful (`numberOfCows` as opposed to `n`). The greater the *scope* where the variable is visible, the *longer* and more *detailed* the name must be (`cows`, `numberOfCows`, `numberOfCowsFromRadioCollars`).

Rule Single letter names can **only** be used for loop control *or* temporary variables.
Example

Bad code!

```
int n = 0;
```

Good code!

```
int numberOfValues = 0;
```

Motivation Single-letter names are confusing except in a context where their context is clear (*e.g.*, as a loop counter. In the bad code example, `n` gives no hint of the variable's *function* in the program. The good code example naming is explicit and will not lead to confusion.

Camel Case Code differs from natural languages: for the ease of processing by a compiler, words separated by whitespace always refer to separate things. Computers cannot group a sequence of words, say "the distance to the sun in millions of kilometers", into a *phrase*.

Rule Identifiers must use camel case (but see clarifications regarding capitalization, below). This requires beginning the identifier with a lower case letter and using an upper case letter for the start of each subsequent word within the identifier.
Example The bad code example removes whitespace but is difficult to read. The good code example uses camel case for enhanced readability.

Bad code!

```
double thedistancetothesuninmillionsofkilometers = 150;
```

Good code!

```
double theDistanceTotheSunInMillionsOfKilometers = 150;
```

Motivation Since whitespace cannot be used within identifiers, words must be connected within an identifier. Underscores add characters to the identifier and are reduce ease of typing. Camel case addresses both readability and typeability.

Capitalization by Part of Program Capitalization is used in alphabetic natural languages to make them easier to read. For example, capital letters in English indicate the beginning of a sentence or highlight a proper noun. Similar (but different) rules apply in other languages. Analogous benefits come from capitalization rules when writing in a programming language.

Camel case is used when constructing a name out of a multi-word phrase. The type of thing being named determines what kind of phrase should be used and the capitalization of the initial character in the name.

Rule Class/interface names must begin with an upper case letter and use camel case after that.
Example The following are examples of good *type* names

```
public class Insect ...
public interface MotorCar ...
public class IngredientCollection {
  class Node {
    ...
  }
  ...
}
```

Motivation Class names should be easy to distinguish from the objects that are created from the class. In Java, a *class* is a new *type* of object. It is a class of things (a blueprint for a type of buildings, for example) that the program will manipulate. The capital letter sets the name of types of objects apart from the names of objects themselves.

Rule Constants must be named using ONLY upper case and underscores to separate words.
Example Constants are the *only* names that use a single case (upper) and underscores to separate words.

```
final double PI = 3.1415;
final int DEBUG_PRINT_LEVEL = 4;
```

Motivation A *constant* is a named value that is used in place of a *literal value*. The constant is named to make it easier to change and, more importantly, to document what the value is used for.
Putting the name of the constant in ALL CAPITALS makes it stand out from variables and types.

Choosing Names Names in a program **must** be meaningful (*e.g.* `numberOfRows` or `rowCount` rather than `n` or `r`). The broader the *scope* (number of lines where the variable is usable), the more detailed (and **longer**) the name must typically be. The following table summarizes the capitalization rules above and gives guidance on picking names for different parts of the program.

Type	Convention	Examples
`class`	Capitalize first letter, `interface` CamelCaseWithin	`class Insect` `interface MotorCar`
Descriptive, singular noun phrase that names a *type* or *family* of values. Singular even if it contains *multiple* values; the class names the type of the container, not contained objects.		
constant	All uppercase, words SEPARATED_BY_UNDERSCORES	`final double PI = 3.1415` `final int DEBUG_PRINT_LEVEL = 4`
Descriptive, singular noun phrase.		

method	lower-case first letter, camelCaseWithin	`void scaleImage(double factor)`
		`boolean isTransparent()`
		`int lengthOfSampleInMeters()`

`void` function is an active verb phrase. `boolean` method is a yes/no question. Otherwise named as the value returned (see variable below).		
variable	lower-case first letter, camelCaseWithin	`double costOfDiamondInEuros` `List<Double> heighsInHands`
Descriptive, singular noun phrase for most variables. Use a *plural* noun phrase if the variable names a collection.		

Rule Spell out words completely in names.
Motivation How much typing do you save if, instead of `gameLevel` you use `gameLvl` or `gameLev`, or (shudder!) `gameL`? How can the future programmer guess which one was used?
Exceptions As with single letter names, short scope can sometimes excuse this. Consistent abbreviations in a program or collection of programs might also make this acceptable. When writing a brand new program, stick with complete words.

Rule Always try to make the **code** clearer with *naming*, *formatting*, or additional *methods* rather than adding lots of in-line comments. Avoid noise comments.
Example
Bad code!

```
// field counts the number of Christmas presents
// stored in this Tree object
private int someThing;
...
sum += i;  // add i to sum
```

Good code!

```
// count all presents under this Tree
private int numberOfPresents;
...
sum += i;
```

The second comment in the bad code *adds nothing*. It repeats, in English, exactly what the Java says. Let the code communicate the steps of the program.
Motivation Think **very** hard about making your variable names describe what is happening. Good names do not take away *all* need for comments but they help make the code easier to read.
Removing comments that duplicate the code is especially important because if you comeback in three months and update the code, you may not even bother to *read* the accompanying comment, let alone update it. Then, in six months, future you will see code and comment that disagree.[5]

Documentation

Documentation encompasses *all* of the human-readable materials that compose and accompany a computer program: variable names, source code comments, file names, README files, and user manuals. These must work together to permit a user of the program to run it and a future programmer to understand, verify, and extend the program.

Naming and source code comments have their own sections above. This section will focus on external documentation, particularly the `README` file you will write for every assignment.

[5] It is said that if code and comments disagree, it is likely that neither is correct.

README

Students **must** turn in a README file with *every* assignment. The README restates the problem the program is to solve, design decisions made by the developer, how to compile and run the program, and a *test plan* on how correct operation was tested for and how the grader can run the same tests.

Rule README is either README.txt (plain text), README.md (GitHub-style *Markdown*), or README.org (*Org mode*) and is in the *root directory* of the material submitted for grading.[6]
Motivation README is intended to go with the source code and be update along with the source code. It is therefore in a format that can easily be typed in a code editor. This also makes sure that it can be read on any given machine, whether or not any given piece of software is installed.

Rule README begins with a restatement of the problem being solved **in the student's own words**.
Motivation Restating the problem in your own words is actually something you should do early in the process of writing a program. If you cannot clearly restate the problem, that indicates that you do not understand it.
Including the restatement in the README makes the directory tree *self-documenting* and complete. It also tells the reader (grader) how you interpreted any parts of it that might be open to that.

Rule README documents *design decisions* made in implementing the solution.
Motivation Did you, the programmer, select any major data structures or algorithms used in the program (that is, were they **not** dictated in the problem statement)? If so, a short statement on the range of possibilities considered and justification for the choice made is in order.
Similarly for any file formats you designed and third-party libraries that you included. This is also a convenient place to give credit for code that you read in solving the problem.

Rule README must include clear, concise instructions on how to *compile* and how to *run* the program.
Motivation The problem statement included how the program interfaces with the world and the solution language has conventions for how to build executables. But tiny variations in interpretation make it hard for the user to quickly run your program. Your README must clearly explain how to run the compiler or build tool, especially if the tool is run from a different folder than the root of the project. Then it must explain how to run the program and how to provide data files and arguments to the running file.

Rule README *documents* the **test plan** used to validate the solution.
Motivation How do you know when *any* computer program is **done**? You stop when the program solves the problem *correctly*. How would you **convince** someone (*i.e.* the grader, your boss, the client) that the program solves the problem and that it is correct? By running tests with known outcomes and comparing expected and actual results.
The assignment may contain some *explicit* test cases. You can include those in the README. More important are the test cases you develop as you figure out what correct behavior is.
You need to give explicit instructions on how to give test input to your program and how to compare the output to that which is expected. These instructions must permit the user to confirm that your tests pass.
The user needs to know: (a) what data to give to the program; (b) how to run the program with the test input; (c) how to evaluate the output for correctness. Of course the programmer had to know those three things in order to test the program in the first place, so this is just a written record of that work.

Git Commit Messages

gitis the *version control* system used in the CS Department. A version control system is a database of the source code that makes up a program, keeping track of how the set of files has changed over time. This permits, at a minimum, long-term undo back to any point saved in the history.

Each project is put into a **git***repository*. Each time you add changes to the repository, you *commit* the changes. Each commit includes a *commit message*. A commit message is documentation.

Rule Each *student* must use a private organization for each *course* to create their repositories for turning in code. The name of the organization is of the form F23-310-laddbc.
Motivation The organization is *private* because only the student and instructor should have access to materials submitted for grading.
The naming of the organization is formatted so the it gives the semester, the course number, and the student's campus computing ID (email w/o potsdam.edu). This makes it possible to write scripts that find all the organizations for a given course offering.

[6] The format of file is indicated by the file extension. Different professors in the department may use or specify different formats and you will want to be familiar with all three.

The organization is owned by the student. The student must create a team with **read-only** access to the repos in the organization and add their instructor to that team.

Exceptions

Rule Each *assignment* is turned in through a private repository in the organization with the naming convention `F23-310-laddbc-p001`.
Motivation The repository is private so other students cannot look at work turned in for grading. The last part of the name, `p001`, for example, is given by the instructor to match their preferred assignment naming strategy.
The naming convention is redundant to make sure that each repository for an assignment in a class will have a unique folder name on the grader's computer.

Rule Each `git` commit message documents what changed at a level of abstraction above the list of changed files.
Motivation `git` can list all the changed files between commits. It cannot automatically document *why* this commit was made. That is what the commit message is for. Especially the first line.

Rule Lines should never be longer than 80 characters.
Rule Never break `import` or `package` lines.
Rule Indentation **must** be consistent in source files. Always use the *space* character for indentation, never the *tab* character.
Rule Use *enough* blank lines to structure your code but do not use *too many* blank lines.
Rule Opening curly braces, `{`, should *never* be alone on a line; closing curly braces, `}`, are alone on a line more often than not.
Rule Sections in a Java source file should be ordered as follows. Skip any empty section.

```
<Package Declaration>

<Header Comment>

<Import Statements>

<Class Declaration>
```

Rule A `.java` file must contain a *single* `class` definition.
Rule Fields/methods in a class should be ordered as follows. Skip any empty section. Within any given section, order alphabetically by the names of the fields or methods. With overloaded methods (including constructors) order by number of parameters.

```
public class <Class Name>
  extends <Parent>
  implements <InterfaceA>, <InterfaceB>, ... {

  <Private Field Declarations>

  <Constructors>

  <Public Methods>

  <Private Methods>
}
```

Rule All header comments must use `JavaDoc` formatting and styling.
Rule **Every** source code file must begin with a *file header comment*
Rule Every method **must** have a header comment with the parameters and return value documented in `JavaDoc` format.
Rule Single letter names can **only** be used for loop control *or* temporary variables.
Rule Identifiers must use camel case (but see clarifications regarding capitalization, below). This requires beginning the identifier with a lower case letter and using an upper case letter for the start of each subsequent word within the identifier.
Rule Class/interface names must begin with an upper case letter and use camel case after that.
Rule Constants must be named using ONLY upper case and underscores to separate words.
Rule Spell out words completely in names.
Rule Always try to make the **code** clearer with *naming*, *formatting*, or additional *methods* rather than adding lots of in-line comments. Avoid noise comments.
Rule `README` is either `README.txt` (plain text), `README.md` (GitHub-style *Markdown*), or `README.org` (*Org mode*) and is in the *root directory* of the material submitted for grading.
Rule `README` begins with a restatement of the problem being solved **in the student's own words**.
Rule `README` begins with a restatement of the problem being solved **in the student's own words**.
Rule `README` documents *design decisions* made in implementing the solution.
Rule `README` must include clear, concise instructions on how to *compile* and how to *run* the program.
Rule `README` *documents* the **test plan** used to validate the solution.
Rule Each *student* must use a private organization for each *course* to create their repositories for turning in code. The name of the organization is of the form `F23-310-laddbc`.
Rule Each *assignment* is turned in through a private repository in the organization with the naming convention `F23-310-laddbc-p001`.
Rule Each `git` commit message documents what changed at a level of abstraction above the list of changed files.