SUNY Potsdam Computer Science Coding Standards

Department of Computer Science

January 6, 2023

1 Introduction

When you learned to write an essay, you learned both the mechanics of writing and how to approach the overall process of writing an essay. This document provides similar guidance for writing a computer program.

1.1 Intended Audience

Before you write an essay, most guides tell you to know your audience. Writing for a precocious ten-year old and writing for a college composition class are both different from writing a cover letter for a job application. The same is true for computer programs.

Computer programs are written for future programmers who must understand the code. This future programmer (who may well be a future you) knows Java and English but needs you to not only solve the assigned problem but clearly explain why the problem needed solving, what your program does, and how your program solves the problem. This explanation is in the form of the Java code, the names you give the things in your code, and the comments that you include.

1.2 Current Department Standard Java Version

Along with standards for code style and commenting, the Department enforces standard versions of the programming languages used in courses. The "foundation language" in our program is Java. The current standard Java version in the CS Department is Java 12. This is the Java version that is installed in the CS Lab (Dunn 302). All student programs submitted for credit must compile under this standard version. It is your responsibility to ensure that you are turning in code that meets the standard, even if you are developing on your own computer.

For courses that do not use Java, details of required language versions and other software tools will be provided by the instructor.

2 Readability

Programmers read a program when joining a team, when validating that the program meets requirements, when requirements change, when evaluating its quality, and while fixing bugs. Programmers read a program over and over again while fixing bugs.

A computer program is read at least an order of magnitude more often than it is written. The emphasis on readability stems from this simple observation.

Much like an essay in a natural language, readability depends on the formatting and structure of the text, the use of spacing and capitalization, and the use of introductions and conclusions at multiple levels (i.e., to the essay, to a section, to a paragraph). These conventions make the English essay much more readable.

Just as an example, consider the previous paragraph without punctuation or word breaks and random line breaks and capitalization[1]:

```
Muchlikeanessay
  INANATURALLANGsUpAaGcEiRnEgADABILITYDEPENDSONTHE
FORMATTINGANDSTRUCTureof t h e t e x t t heuseofconventional
  spacingandcapita l i z a  tionspacin
   gandtheuSEOFintrudo  ctionsandin clusionsat
  multiplelevelsietotheessaytoasectiontoaparagraph
tHeseconventionsmaketheenglishspeacsinsg aymuchmorereadable
```

This section has an overview of the structure of a Java source file followed by subsections on documentation and formatting.

[1]The unformatted paragraph is not only difficult to read. It is also difficult to debug: several spelling errors and minor variations from the original were introduced and are non-obvious without careful scrutiny.

2.1 Source Files/Class Declaration

A source file is the `.java` file that is named for the **single** top-level **`class`** declared inside it. The sections of the source file should be separated by exactly **two (2) blank lines**; any empty section can be omitted along with the blank lines for that section.

```
<Header Comment>


<Import Statements>


<Class Declaration>
```

2.1.1 Header Comment

Every file submitted must have a Javadoc header comment. A Javadoc comment begins with a slash followed by **two (2)** stars (see following example).

The header comment has three parts: class summary, detailed class description, and the programmer's identification block.

```
/**
 * 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
 */
```

A file header comment begins with a single-line **summary** of the class defined in the file. Do not just repeat the name of the class in English. Give enough detail so another programmer can figure out if this is where they will find whatever they are looking for. (Actually, the summary permits a programmer to know that what they are looking for is **not** in this file. If the file cannot be dismissed instantly, then the programmer must read the detailed description.)

After a blank line in the comment comes a deeper **description** of why this class is part of the program. It gives usage details and how this class connects to other classes in the program. References to sources for the code in the file belong here.

The end of every file header comment identifies the programmer and project to which the file belongs. Get in the habit of including this block in every `.java` file you create: no **program** will be graded if any file that makes it up is missing the id block.

(Hint: Put the id block in each file when you create it. Or teach your editor/IDE to include it whenever you start a file.)

The lines begin with `@` and a keyword because they are designed to be processed automatically by software tools like `javadoc`.

2.1.2 Import Statements

The import statements list the classes that this class declaration depends on. Required classes are imported by giving their full names after the **`import`** keyword.

A class's full (or qualified) class name is a dot-separated name, a lot like a Web address. Using an asterisk as the last part of the name will import **all** classes that share the specified first part. For example

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

will import **all** classes defined inside the `java.util` package. When writing (or reading) an import with a wildcard (the star matches **any** name in the package, so it is a wildcard), you do not know which, nor how many, classes are imported.

If you include the full name in the **`import`**, then you know exactly what is being imported. For example, if you were using `List`, `Map`, `ArrayList` and `HashMap` classes in your code:

```
import java.util.ArrayList;
import java.util.HashMap;
import java.util.List;
import java.util.Map;
```

Always prefer importing specific classes over the use of wildcards. The first example above should be considered bad code:

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

Imports should be ordered in alphabetic order (unless there is an explicit ordering constraint on content; Java should not need this). Each import is to be on a single line: it is not to break to the next line, even if it goes past 80 characters.

2.1.3 Class Declaration

There is to be exactly one top-level class declaration in a `.java` source file. The class declaration appears on one line unless it has an **`implements`** or **`extends`** clause. Those clauses begin on a new, indented line. As in

```java
class Gargoyle
  implements Comparable {
  ...
}
```

Field Declaration Order The fields of a class all come before the methods. All **`static`** fields come before all object-level fields. Constant fields (and constant, static fields) should come at the beginning of the section for that type of field.

Method Declaration Order A standard order for methods declared inside a class is like using call numbers to sort books in a library: the reader knows where to look and can tell if a section is empty.

`main` The public, static, void `main` method is the first method, if it is present.

Constructors All of the constructors (if there are any) follow `main`.

Overloaded Methods Methods with the same name (overloading the same name) must be adjacent in the source code.

2.1.4 A Deeper Look

Larger, more complicated programs require additional levels of abstraction. In particular they need multiple Java packages. A package is a collection of source files in a directory named for the package; each source file must also include a line declaring the package to which it belongs. The package declaration is a new, first, section in the source file:

```
<Package Declaration>



<Header Comment>



<Import Statements>



<Class Declaration>
```

Package Declaration If the `.java` file appears in a non-default package, then the declaration of the package name is the first line in the source file. It is all to be on one line: it is not to break to the next line, even if it goes past 80 characters.

Static Import Statements **`import`** statements come in two flavors: regular (non-static) imports and **`static`** imports. Regular imports should come before **`static`** imports and the two groups should be separated by a blank line.

As with regular imports: always prefer specific, non-wildcard **`static`** imports; the group of **`static`** imports should be sorted in ascending alphabetic order; each **`static`** import is to be on a single line, even if it goes past 80 characters.

2.2 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 should all work together to permit a user of the program to run it and another programmer to understand, verify, and extend the program.

2.2.1 Naming

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`).

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

Spell out words completely: it is hard to remember how you abbreviated the word **level** the next time you work on the code: was it `lvl`? `lv`? `lev`? Surely not just `l`?

Type	Convention	Examples

`class`	Capitalize first letter,
	CamelCaseWithin
		`class` `Insect`
		`class` `MotorCar`
		`class` `IngredientCollection`

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; plural noun phrase if the variable names a collection.		

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 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 kilometers", into a phrase.

Programmers jam all the words together: `thedistancetothesuninkilometers`. Now the phrase is a single **word**[2], a very difficult-to-read[3] word. There are two standard fixes for this: underscores and camel case.

i Replace spaces with underscore characters ("_") instead of erasing them. The resulting phrase would have no spaces (for the computer) but visual spacing between the words (for humans): `the_distance_to_the_sun_in_kilometers`. Improved readability comes at the expense of degraded writablity (or at least typability).

ii Camel case removes the spaces as in the first option, capitalizing the initial letter of each word to visually separate them: `theDistanceToTheSunInKilometers`[4]. The capitalization of the initial character in the name depends on what kind of thing is being named (as discussed below).

Camel case is the standard way of combining multiple words into a single name in Java.

Capitalization by Part of Program 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.

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. Class names must begin with a capital letter and use camel case after that. The capital letter sets the name of types of objects apart from the names of objects themselves.

The class name should be a singular noun or a noun phrase: `Warehouse`, `SkiLodge`, `ModernOfficeComplex`.

[2]See Finnish for natural language examples of the same.

[3]In written English, we can use dashes for much the same purpose, to make a phrase into a single adjective.

[4]The name camel case comes from the interior capital letters looking like the hump(s) of a dromedary.

Constant 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 (finding all the 4's in a program is possible; which ones are being used to indicate the number of F franc to the US dollar?) and, more importantly, to document what the value is used for.

Constants are the only names that use a single case (upper) and underscores to separate words.

The constant name is a noun or a noun phrase: `STATE_COUNT`, `AVOGADROS_NUMBER`, `HASH_TABLE_SIZE`, `FRANC_PER_DOLLAR`.

Method A method (or a function) is a named block of code that adds a new command to the language. It starts with a lower-case letter and uses camel case.

The return type determines how the method is named:

`void` A specific, active verb phrase. The method is called to do something: `move`, `playSoundEffect`, `setTaxRate`.

`boolean` A yes/no question that the method answers: `isAVampire`, `canMove`, `validCoordinate`.

Other Describe the value returned by the method as a variable, perhaps with a description of how the parameters are used: `indexOfSmallestEntry`, `celsiusFromFahrenheit`, `getTaxRate`.

Variable A variable (or a field (in a class but not within a method) or a parameter (in the parameter list of a method)) begins with a lower-case letter and uses camel case.

A variable is a thing: a singular noun phrase is typical as in `zipCode`, `totalPoints`, `distanceFromEarthKilometers`.

Sometimes a variable names a collection of values: a plural noun phrase makes sense as in `allZipCodes`, `teamScores`, `fleetSpaceShips`.

2.2.2 Comments

Comments are the most important documentation for programmers reading the code in the future. They serve as a sort of an outline to find which section of the code performs a function without having to keep track of all the details in the code itself.

Good commenting conventions address the four I's:

Introduction WHY? Why is this file part of the solution. At a high level, what routines are in the current file?

Identification WHO? Name the programmer responsible for this code. Gives the author credit (or blame).

Intent WHAT? What does this method do? The next level of detail from the name.

Implementation HOW? How does the code work? One level up from the code, guiding the next programmer to see how the statements of Java go together.

File Header Comment A file header comment (as discussed earlier) has three parts: summary, detailed description, and identification block.

Together the summary and detailed description introduce the class in the file, addressing the intent of the class it provides. The programmer's id block identifies the author of the code in the file.

```
/**
 * 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
 */
```

The **summary** does not just repeat the name of the class (and file) in English. It is a single line that permits a programmer to quickly determine whether or not they need to read the detailed description. When searching for the part of the code that does some specific thing, the quicker one can stop reading unrelated code the better.

The **description** is detailed in comparison with the summary. It is also a statement of the intent of the whole file, a description for an outsider using the class to know what it is supposed to do. Mention any general assumptions made in the class, non-library classes upon which this class depends, and references to any sources from which you derived code.

As mentioned earlier, the **id block** uses keywords starting with the ampersand (`@`) because these are treated specially by the `javadoc`[5] program. This id block is required in all CS classes. Files without an id block will not be graded and given a 0 (zero).

Method Header Comment Every method in every class must have its own Javadoc header comment. The comment begins with the slash and two asterisks (makes it a Javadoc comment) and, like the class header comment, has a **summary** and **detailed description**. The difference is in level: a method header is about the intent of the method, answering the questions "What does this method do?" and "How is this method used?" (What are the parameters and what do they mean?).

```
/**
 *  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 {...}
```

[5]`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.

The **summary** is the one line intent of the method[6] The **description** is a more detailed description of the intent, including use of parameters.

The **keywords** are to 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. Obiously: not all keywords are used in every header.

`@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.

`@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.

Field (variable) Declaration Comments When declaring a field (read the or variable throughout this section; hopefully variable scope and good naming practices will mean this primarily applies to fields), sometimes you want to tell the future programmer something about the field being declared. You start to write a comment on the declaration.

[6]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.

Stop!

Think very hard whether or not you can use the name of the field to communicate the same thing. Always prefer having the name of the field document the field to having a separate comment on the declaration. Using the name, the important information is available everywhere the field is used. The future programmer cannot forget to read the comment. The field name must describe its function/purpose in the program. This is known as self-documenting naming.

If you beleive that comments are unavoidable, put comments on the line(s) before the declaration. Use a `//` (end-of-line) comment if the comment is a single line. Use `/**/` comments otherwise.

In-line Comments Code, like fields above, should be clear and named methods, variables, and constants should explain what is happening. This is, of course, not always possible. So, strategically placed comments in the code serve as landmarks for the future programmer.

Put comments on the line(s) before the code being documented. Use a `//` (end-of-line) comment if the comment is a single line. Use `/**/` comments otherwise. In-line comments **must** appear on a line or lines by themselves. Not this

```
sum += data[i]; // add value to the sum
```

but rather this

```
// add value to the sum
sum += data[i];
```

The only exception to the "no comments after code" rule is when you are documenting the closing curly brace of a block, including a comment on the **else** line of a selection statement:

```
if (value >= 0) {
 . . .
} else { // value < 0
 . . .
}

while (!inputString.equals(QUIT_COMMAND)) {
 . . .
} // while (inputString.equals(QUIT_COMMAND))
```

Example Comments The following examples show how to apply these rules, along with good naming and formatting, to produce readable code. The class containing these methods is left out for brevity.

```java
/**
 * Take the square root of the parameter.
 *
 * @pre nonNegative >= 0
 * @param nonNegative the (non-negative) double to square root
 * @return square root of nonNegative
 */
public double getSquareRoot(double nonNegative) {
  return Math.sqrt(nonNegative);
}
```

```java
/**
 * Calculate the sum of the values in the array.
 *
 * Sum the values in the array; returns zero if there
 * are no elements.
 *
 * @param valuesToSum collection (array) of double to sum
 * @return the sum of all values; zero (0) if the array is empty
 */
public double sumOfValues(double [] valuesToSum) {
  double sum = 0;
  for (int i = 0; i < valuesToSum.length; i++)
    sum += valuesToSum[i];
  return sum;
}
```

```java
/**
 * Print the sum of the values in the array to standard output.
 *
 * If the values array is not empty, use sumOfValues to calculate
 * the sum of the contents and then print it.
 *
 * @pre values.length > 0 (values is NOT empty)
 * @post the sum of the values is printed to standard output
 * @param nameOfTheArray the name of the array in printed message
 * @param theArray array of values to print the sum of
 * @throws IllegalArgumentException if theArray is empty
 */
public void printSum(String namOfTheArray, double [] theArray)
  throws IllegalArgumentException {
  if (theArray.length == 0)
    throw new IllegalArgumentException("theArray is empty");

  double sum = sumOfValues(theArray);
  System.out.println("The sum of " + nameOfTheArray + " = " + sum);
}
```

2.2.3 README

With every programming assignment, students must turn in a README file. The README documents the overall problem being solved, how the program was tested for correctness, and how to run the program.

Format The file can be in plain text (README.txt), GitHub-style Markdown (README.md), or Org mode (README.org).[7] The most important factor of the format of the README is that it be easily read in a plain text (**read:** code) editor.

Problem Restatement Restating the problem that you are solving has several benefits: it documents your understanding of the assignment, it helps the grader, and it decouples your solution from the URL where the assignment lived. Documenting your understanding of the assignment is especially useful early in the solution process; if, after reading the assignment, you try to restate what you're doing, you will develop questions for the instructor on the parts you cannot restate.

The grader benefits from your restatement in both seeing what you did or did not understand about the assignment (see above) and is reminded

[7]The format of README should be indicated by the file extension. Your professor may specify which format(s) to use in a given course so you should familiarize yourself with all three types.

what the assignment was about just in case it has been a while since it was assigned (hard to imagine Dr Ladd getting behind in grading, I am sure).

The problem, restated in the solution's README makes the solution complete; it does not depend on the assignment remaining available into the future. The solution fits better in a programming portfolio when it is self-contained.

Testing Criteria Consider any computer programming project: how do you know when it is **done**? An important concept in software engineering is defining what it means for code to be done. One dimension of that is, typically, for the code to be correct. How could you convince someone (in particular a grader) that your code is correct?

With an assignment there are often some explicit test cases, examples of what the code should do in some cases. Of course, early in their programming career, most students realize that the grader has a broader collection of implicit test cases, cases that follow from the requirements of the software. Students do well to think about these additional test cases.

As projects become more advanced, the number of implicit requirements is greater than those explicitly described in the assignment. Part of programming is being able to turn requirements into acceptance tests. Your acceptance testing is part of your solution; you must document it just as you would any other part.

The **most important** part of this documentation is to explain what the expected results are for a given set of input. The reader (grader) 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.

Execution Instructions The grader needs to be able to compile and run the solution. It is possible the assignment explained how the code should interface with the world; it is likely that some student(s) interpreted that interface differently than the instructor who wrote it. You **must** explain, clearly, how the solution is compiled (what tool or tools are needed, what folder are they to be executed in, what are their parameters and what do those parameters mean) and how it is run (what is the command-line one types in the shell, what do the arguments mean, what limitations are there on input values and input locations).

2.3 Formatting

Formatting, the layout of the text inside source files, is used to break up the code into logical units.

- The sections of a source file are separated by exactly two (2) blank lines (see Section 2.1, Source Files, for order and details on source file sections).
- Method ordering: `main` first, all constructors next.
- All methods with the same name (overloads) must be adjacent (and constructors come before all methods but `main`).
- No more than one statement per line.
- Line length should not exceed 80 characters.
  Exceptions: **`package`** declaration; **`import`** statements.
- Use **only** spaces for indentation (see Section 2.3.1, Whitespace, for more details).
- Indentation is either 2 spaces per level or 4 spaces per level (your professor may specify). Whichever indentation is chosen, indentation must be **consistent** across all files turned in together.
- No line should ever begin with an opening curly brace, `{`.
- The closing curly brace, `}`, should align with the indentation of the line ending with the matching opening curly brace, `{`.

2.3.1 Whitespace

Whitespace, the blank space on a printed page, is something that is not there when humans read it. The computer must keep track of all the characters including blanks and the ends of lines.

Indentation uses **only** spaces for indentation.[8] Blank lines should be empty (no spaces or other non-printing characters).

In particular: configure your code editor to use spaces and **not** the tab character for indentation.

Whitespace in source files is to use **spaces** and **end-of-line** characters. In particular, turn off the use of tab characters for indentation in your editor and use the escaped character sequence `'\t'` for tab in strings.

Each source code and documentation file should end at the beginning of a blank line (the last character is an end-of-line).

Again: Indentation must use only spaces.

[8] `Makefile` formatting requires the use of the tab (`'t'`) character. It is the thereby excepted from this rule.