

Coding Style

Most of this handout was written by Nick Parlante and Eric Roberts, then edited for our own clandestine purposes.

When writing a paper, you can have well-crafted, correctly spelled sentences and create “A” work. Or you can hack out the text in a hurry. It will not look as good, but it can convey your thoughts and get the job done; it’s worth maybe a “B” or a “C”. Computer code is not like that. Code that is messy tends to have all sorts of bugs and other problems. Messy code attracts problems like a half-eaten lollipop attracts lint (and that’s never pleasant). The problems and bugs in poorly written code tend to compound each other and pile up, so the code ends up being nearly worthless. It has bugs. Nobody knows how to fix them or add features without creating more bugs. Once code is in that state, it is hard to escape. In a sense, code tends to be more either “A”, or “D” or “F”. Therefore, it is best to write code that is clean to start, and keep it clean as you add features. This is one of the lessons in CS for successfully building large projects. For that reason CS 106A emphasizes the habits of clean, well-engineered code right from the start, building the right habits for the future.

One reason to write clean, well-structured code is that it works better and takes less time in the end. The other reason to write clean code is that it is just more satisfying to do a good job on something. Clear, elegant code feels right, just like any other engineering or artistic creation.

The messy code trap

It is a basic fact of computer science that poorly designed, messy code is harder to build and debug than clean code. It is tempting to just type out a quick solution as it occurs to you to get started. It is better to take a little more time at the start to build the clean version, creating fewer headaches in the debugging phase. Once code gets messy, it’s hard to clean it up. It’s easier to start clean, and then keep it clean with each addition. The worst possible strategy is to build the messy version, do your debugging on that, and then clean it up before turning it in—all the work and little of the benefit! Do it the right way from the start, and you’ll be happier.

Decomposition

Decomposition does not mean taking a completed program and then breaking up large methods into smaller ones merely to appease your section leader. Decomposition is the most valuable tool you have for tackling complex problems. It is much easier to design, implement, and debug small functional units in isolation than to attempt to do so with a much larger chunk of code. Remember that writing a program first and decomposing after the fact is not only difficult, but prone to producing poor results. You should decompose the *problem*, and write the program from that already decomposed framework. In other words, you are aiming to **decompose problems, not programs!**

The decomposition should be logical and readable. A reader shouldn't need to twist her head around to follow how the program works. Sensible breakdown into modular units and good naming conventions are essential. Methods should be short and to the point.

Strive to design methods that are general enough for a variety of situations and achieve specifics through use of parameters. This will help you avoid redundant methods—sometimes the implementation of two or more methods can be sensibly unified into one general method, resulting in less code to develop, comment, maintain, and debug. Avoid repeated code. Even a handful of lines repeated is worth breaking out into a helper method called in both situations.

Readable code

One metric for good code is that it “reads” nicely—that someone sweeping their eye over the code can see the algorithmic idea at hand. The original programmer had an idea in mind—a way to solve the problem. Does the code communicate that idea? Writing readable code is important both because it will help any future reader and because it helps you avoid your own bugs. Bugs, after all, are simply where the code expresses an idea, but it is not the idea you had in mind. Readable code has fewer bugs.

Variable names

The first step in readable code is choosing good names for variables. Typically a variable gets a noun name that reflects what it stores—`width` or `height` or `bankBalance`. If you have the number 2, but do not know anything about it, then the generic `num` is an okay name. If you know more specifically that it's a weight or a number of pixels then the name should reflect that knowledge. In Java, the convention is to begin variables with the first word lowercase, and uppercase later words like this: `bestScore`, `remainingCreamPuffs`. If you have a pointer to an object but without any more specific word to use for its variable name, then you can use the name of the class in lowercase. So if code deals with a `Circle` or `Person` object, then obvious variable names are `circle` or `person`. If you know something more specific about the objects, then more specific names like `leftCircle` or `mother` are better. There are a few idiomatic one-letter names—`i`, `j`, `k` for `int` loop counters; `x`, `y`, `z` for coordinates. These are in such wide use that they make very readable code just by familiarity.

Method names

If variables names are the nouns, method names are the verbs. Method names should reflect the action they perform—`removeAll()`, `drawLine()`, `getX()`. The prefixes `get` and `set` have a typical role. A `get` method gets a piece of information from an object, either a value that the object stores or computes: `getWidth()`, `getNumChildren()`. Likewise, `set` methods typically are used to pass a value in to an object for it to store or use: `setWidth(int width)`. Methods that return a boolean (i.e., predicate methods) are often named starting with `is` or `has`.

Whitespace

Use whitespace to help separate the logical parts of the code, in much the same way that paragraphs separate groups of sentences. Rather than write a block of 20 lines, it's nice to

put in blank lines to separate the code into its natural 6-line sections that accomplish logical sub-parts of the computation. Each little section of code might have a comment to describe what it accomplishes. Likewise, you can use whitespace to show the logical grouping of elements within a line. Do not run everything together with no spaces. Here are a few examples

```

/* many terms with no spaces -- never do this */
int i=2*i+12/i;

/* spaces around every operator—okay */
int i = 2 * i + 12 / i;

/* could add parens for readability */
int i = (2 * i) + (12 / i);

/* here's the same idea, but with boolean expressions... */

/* spaces - ok */
if (i * 12 < j) {

/* could add parens for clarity */
if ((i * 12) < j) {

```

Indentation

All programming languages use indentation to show which parts of the code are owned or controlled by other parts. In CS 106A, whenever there is a `{`, the code on the next line should be indented—this applies to methods, classes, `if`-statements, loops, and so on. Eclipse will do this automatically. Hit the tab key to indent one level manually. You can also select a few lines and use tab to move them all right on level, and shift-tab to move them all left one level. At the end of the indented code the matching `}` should not be indented. In this way, the indented section is visually set-off from the outer `{ }` that controls it, as shown:

```

if (i > 10) {
    println("i too big");
    i = i % 10;
    someMethod(i);
}

```

Comments

Comments add the human context to the raw lines of code. They explain the overall flow and strategy of what is going on. Comments point out assumptions or issues that affect a part of the program that are not obvious from the code itself.

As you write larger and more complex pieces of code, comments help you keep track of your own assumptions and ideas as you are building and testing various parts of the code. There gets to be more than you can keep in your head at one time. The first step is good variable and method names. They make the code “read” well on its own, so fewer comments are required.

Class comments

Each class should have a comment summarizing what it does. Typically the class comment will mention what sort of data the class encapsulates and what sort of methods it implements. Professional quality documentation for a class or group of classes intended for use by others, such as the `String` class, will also have a few introductory paragraphs of discussion of what sort of problems the class solves and what typical client use of the class looks like. For a system of classes, there may be an architectural overview that summarizes the role of each class and how they all fit together to build the program.

Variable comments

Sometimes the meaning of an instance variable or local variable is completely clear just from its name. For a complex variable, there is often extra contextual information about the variable that the code must be consistent about. A comment where the instance variable is declared is the perfect place to document such side-issues for the variable: what are its units? Are there constraints on what values it is allowed to take on? For example, `weight` might be the perfect name for an instance variable indicating the weight of the object, but you still need to know, say for a car simulator, that it is in pounds, or that the weight is for the car but does not include the passengers or fuel. There is often ancillary information about an instance variable—its meaning, assumptions, and constraints—beyond what is captured in its name. The comment for an instance variable can capture this extra information about the variable in one place.

Method comments

Method comments should describe what the method accomplishes. Emphasize what the method does for the caller, not how it is implemented. The comment should describe what the method does to the receiver object, adding in the role of any parameters. In the standard comment style used with `Javadoc`, the method comment begin with a verb in the third-person singular form (typically ending in “s”) describing what the method does. For a complex method, the comment can address the preconditions that should be true before the method is called, and the postconditions that will be true after it is done.

An example of method commenting is shown on the next page.

```
/* Method: removeAll */
/**
 * Removes all the shapes from the window.
 */
public void removeAll() { . . . }

/* Method: addBet */
/**
 * Adds the given bet amount to the pool.
 */
public void addBet(int bet) { . . . }

/* Method: getCollisionOdds */
/**
 * Returns the odds between 0 and 1 of a collision
 * with the specified other craft.
 */
public double getCollisionOdds(Craft other) { . . . }

/* Method: addCreature */
/**
 * Adds the given creature to the room on which this
 * method is invoked. The creature must be valid before
 * this method is called. After the call, the Creature
 * object will have been updated to show its location as
 * the room into which it was placed.
 */
public void addCreature(Creature creature) { . . . }
```

Attribution

All code copied from books, handouts or other sources, and any assistance received from other students, section leaders, fairy godmothers, etc. must be cited. We consider this an important tenet of academic integrity. For example,

```
/**
 * isLeapYear is adapted from Eric Roberts' text,
 * The Art and Science of Java, p. 106.
 */
```

OR

```
/**
 * I received help designing the decomposition of Breakout, in
 * particular, the idea having a method to handle one ball in play,
 * from Ben Newman on Monday, Oct. 22, 2007.
 */
```