OSU Style Rules for Good Code

* Break code up into logical methods. Don’t put everything in main()!
* A method should do exactly what the contract says it does *and nothing else*.
* Within methods, use blank lines to break things up into sections. No walls of code!
* Give variables and methods logical names that make it easy to understand their purpose. There are only a few situations where one-letter variable names are acceptable.
* There is no award for being able to cram the most operations into one line.
* Avoid break and continue statements in loops. Do not have multiple return statements in a method (or any return statements in a void method).
* With a few exceptions, always assign numeric values to variables (ringsForTheElvenKingsUnderTheSky = 3) rather than simply using them directly in calculations (totalRings = 3 + 7 + 9 + 1) (these are sometimes called “magic numbers” because they appear like magic with no explanation).
* Write comments when needed to explain what your code is doing.
  + BUT! Don’t write comments to explain things that should be obvious to anyone who knows Java. Using the above style rules to make code clear to begin with is MUCH better than writing messy code and then using a bunch of comments to explain it. When in doubt, add a comment, but if you follow the rules listed here, you shouldn’t need a lot.

Why do all this, you ask? Read on.

How iz kode formed?

For most of the projects in this course, you will be provided with a “skeleton code” to get you started. You won’t be beginning with an empty file; you’ll be filling in the blanks. You might think this is “unrealistic,” or that “no one in the real world would write code like this.”

Actually, very little “real world” coding starts from a blank slate. You’re usually adding on to or modifying existing code. You could be fixing a bug, or a customer could want a new feature added. Even if you’re creating a “brand-new” service, you’re probably going to be borrowing heavily from existing code. It saves a lot of time and is much more likely to work.

What this means is that code must be *easily adaptable*. If you follow the principles of style laid out in this course, it will be.

Another important thing to keep in mind is that if you work in a reasonably large company, you will have to deal with code that was written by someone you never met, who might not be at the company anymore, who is too busy to respond to a bunch of “How does this work?” emails. Oh, and when I say “someone,” I mean “3-5 different people.”

So the other thing that code absolutely must be is *easily* *understandable*.

Writing code that is understandable and adaptable is fundamental to becoming a good programmer. But that may not be the easiest thing in the world to appreciate when you’re working on projects for this class.

Adaptability? You’re never going to look at this code again after you turn it in! (NOTE: we do not recommend this. Project content can and will come up on exams.)

Understandability? The main people who need to understand your code are the graders, who have access to the project description and have already done these exact same projects themselves. They have a pretty darn good idea of what’s supposed to be going on.

So Why Bother?

Here are two good reasons why the style rules are important (this is *not* an exhaustive list):

1. Humans are habit-forming creatures. If we do something once and it doesn’t kill us or electrocute us, we’re probably going to do it again. Writing code using bad practices is no different.

SURGEON GENERAL’S WARNING: Bad coding practices are habit-forming!

2. These rules sound simple, but you don’t know how to follow them until you sit down and try. What spacing makes your code easiest to read? What’s the most logical way to break your code up into methods?

And would you rather be answering these questions for the first time in a classroom, or on the job (or in a job interview)?

But Mah Code WORKS!!!

“Ben, I tested it really thoroughly, and it works perfectly every time, and it does exactly what it’s supposed to do, so I think I should get full points kthxbye.”

First: there’s a good chance you didn’t test it as thoroughly as you thought you did! Frequently there’s something you never thought to try that could break your code. This is true whether your name is Linus Torvalds, Ben Trevor, or thisClass.getRandomStudentName().

Second: just being able to write code that works isn’t that impressive. My *grandma* can write code that works (not even joking about that). Writing *good* code, not just *working* code, is what will distinguish you from the other coders out there.

Remember: basic coding ability is a prerequisite for this course, so you should already know how to write code that works. In this class we’re looking for something more.

Efficiency

Efficiency is good! Faster is (almost) always better. But there are times when efficiency is a top priority, and there are times when it’s a secondary concern.

In this course, we aren’t as worried about efficiency as we are about style. Every piece of code you write for this class, with one or two exceptions that will be pointed out to you, should terminate in less than ten seconds. If it takes longer than that, then there’s an infinite loop or a serious inefficiency that you need to fix. Otherwise, just watch out for obvious inefficiencies, like pointless calculations or needlessly repetitive code.

Bottom line: if you have a way to make your code slightly more efficient but the change makes it less readable or less well organized, don’t do it.

Final Thought: There is no Perfect Style

At OSU, we promote a style that leads to adaptable, understandable code. Many elements of this style are used at other places, for the same reasons. Some places, on the other hand, may disagree with part or even all of it.

But no matter where you go, *they will have a style*, a particular way they want you to write code (and if they don’t, well . . . maybe think twice about going there). The ability to learn and follow a style is one you will need. So even though your future employer probably won’t copy OSU style exactly, learning and using it is still a valuable experience. Hooray!