Java Style Guidelines
CS201: Data Structures
Anna Rafferty

As you look at examples in the book and in class, you'll develop a better and better sense of good style. Good style encompasses things like how to design your classes, how to name your variables, how to organize your methods within class, and even how to structure your code within a method. Companies often have style guidelines to make their code more maintainable; for example, here are Google's guidelines for Java. This document has some pointers for how we'll do things in CS201. Note that many of these things are not required by the language, but they follow particular conventions of Java programming (e.g., class names start with a capital letter) and will make it easier for all of us to quickly understand one another's code. This list is certainly not an exhaustive list of good style guidelines, but it should provide you with a starting point.

- Capitalization: Class names should start with a capital letter (e.g., `String`). Variable names should start with a lower case letter (e.g., `myString`). If something is multiple words, use "camel case": word boundaries are marked by capital letters (e.g., `myVeryLongVariableName`). If you're declaring a constant, you can use all caps (e.g., `public static final double PI = 3.14159;`).

- File organization: Put import statements at the very top of your file. Put Javadoc comments (comments that start `/**`) about the class/interface between the import statements and the declaration of the class/interface. Put fields (both static fields and instance variables) before any constructors or methods. Put constructors before any methods. Put `main` (if you have it) after all other methods.

- Comments: Include comments summarizing the whole class, and a comment for each method describing what that method does. Use comments to indicate the purpose of a class's fields. All of these types of comments should be Javadoc style comments. When necessary, include comments within methods to describe the logic of what you're doing. Sometimes, it may be easy enough to just read the method without including comments inside of it. Think: if another person who understood Java read this, would it be clear without a comment? Comments should focus on the logic (e.g., a comment before in the body of an if block saying "Handles positive number case by printing each step in the counting down") rather than just repeating what the syntax does (e.g. "increments i "). When writing a single comment line, use `//` rather than `/* */`.

- Names: Use descriptive variable and method names! In for loops, things like `i` or `j` for the incrementer are fine. Creative names are useful to the extent that they help you and others understand your code - they aren't helpful if they obscure the meaning.

- Capitalizing names: All variable names and method names should start with a lowercase letter, with the exception of constant variables. Constant variables should be all uppercase. All class names should start with an uppercase letter.

- Multiword names: All multiword names should use camel case, where subsequent words in the name start with a capital letter (e.g., `myAwesomeMethod`). The exception is constant variables, which should use underscores (e.g., `MAXIMUM_SIZE`).

- Line length: Try to keep your lines under 80 characters for readability.

- Indentation: Indent just like you did for Python - bodies of methods, bodies of loops, etc.

- Commented out code: When you turn in an assignment, don't include any code that you've commented out (e.g., stray code that didn't work or from an old implementation), unless the assignment explicitly tells you to do so.

- Braces: Always include braces around the body of if/else/for/while/etc., even if the body is only one line. Put your opening brace on the same line as if/else/for/while/etc. and the closing brace on a line by itself (or for conditionals, on the same line as the else if/else for the next case), e.g.:

```
if (line != null) {
    System.out.println(line);
}
```

- Whitespace: Follow standard rules for whitespace around punctuation. That is, do not use whitespace before a comma, semicolon, or colon. Do use whitespace after a comma, semicolon, or colon except at the end of the line. Surround binary operators with a single space on either side for assignment (=), comparisons (==, <, >, !=, <=, >=), and Booleans (&&, ||). Not (!) can be placed directly in front of a variable or method without a space after it.

Structuring your code

The above guidelines focus a lot on presentation, such as how to name things to make it easy for an experienced coder to quickly skim your code and understand it. As you develop your understanding of coding style, you can also start focusing on proper abstraction and control flow. Think about how to design your code to try to avoid unnecessarily complex code (do you really need a bunch of cases or does the final case cover all the special cases?) and to avoid duplicating the same code in multiple places. Coding with good style means writing your code in a way that's more easily maintainable, testable, and readable by others. Simplifying your code often helps with all of these goals: it means that if you find a bug, you only have to fix it in one place, and there are fewer parts to test overall. Here are some tips on good structure:

- Class design: Think carefully about whether fields should be public or private, and don't make them public without a good reason. When deciding whether to make a variable an instance variable or a local variable, think about where and how it will be used. Does the variable refer to persistent state? Probably it should be an instance variable. Will the variable only be used by a single method and will its value always be reset at the beginning of that method? Probably it should be a local variable. Instance variables are handy for storing state, but they can also make your code hard to maintain if there are too many of them and it's not clear why each one exists. Be cautious in adding instance variables - make sure there's a good reason. Similarly, think through whether your fields should be static or non-static.

- Use methods to abstract out shared functionality: Rather than copying and pasting code into multiple place, write a method that can be called multiple times. To keep each of your methods easy to understand and test, abstract out different parts into their own methods. Writing lots of small methods, some of which call others, makes it easier to get started on programming (you only have to make a little bit work!). Building a larger method that calls smaller methods makes it easier to test the larger methods: you first confirm through testing that each of the smaller methods work properly, and then you only need to establish that the larger methods brings together these different parts correctly. If you're writing methods that should only be used by other methods within the class, make them `private` to hide them from users of your class.

- Conditional structure:

  - When a series of if-statements are exclusive, write a sequence of if-else if (-else) statements rather than a series of if-statements. This helps communicate to others that the conditions are exclusive, rather than them having to discover it by deeply understanding your code.
  - Sometimes you have the choice between nesting if-statements and writing one if-statement that checks a boolean condition. For instance the two functions below have the same functionality:

```
public static void sample1(int myNumber) {
    if (myNumber != 0) {
        if (myNumber / 2 > 111) {
            System.out.println("Big!");
        }
    }
}

public static void sample2(int myNumber) {
    if (myNumber != 0 && myNumber / 2 > 111) {
        System.out.println("Big!");
    }
}
```

The second one is preferred because it makes it easier to see the whole condition at once.

- Working with booleans (true/false values): Here are two different code samples with the same functionality:

```
public static boolean boolVersion1(myNumber) {
    if (myNumber > 3) {
        return true;
    } else {
        return false;
    }
}

public static boolean boolVersion2(myNumber) {
    return myNumber > 3;
}
```

The second version is preferred as it simplifies the code.

- Working with booleans (part 2): Here are two different code samples with the same functionality:

```
public static void boolVersion1(myBoolean) {
    if (myBoolean == true) {
        print("You passed in true!")
    }
}

public static void boolVersion2(myBoolean) {
    if (myBoolean) {
        print("You passed in true!")
    }
}
```

The second version is preferred because there's no need to explicitly check a boolean value against true (or false - use `!` instead).

Please come talk to me if you have questions, and happy coding!