1.1 Introduction

This is a program hygiene and best practices guide for CS314. It contains a list of rules we expect you to meet for the code you turn in for programming assignments. This page covers the entire semester, so it is not meant to be something that you memorize and understand entirely through one read through. Rather, it is a reference guide that you will be able to use to look up rules and examples.

Note that this guide is not 100% comprehensive. A guide that tried to cover every particular issue would be ten times the size of this one. **There may be specific things on each assignment we are looking for and the TAs and other graders may find issues not covered by this guide. If so they will provide feedback so you can improve the code you write.**

1.2 Motivation

"The cleaner and nicer the program, the faster it's going to run. And if it doesn't, it'll be easy to make it fast." - Joshua Bloch

"Code is read more often than it is written. Code should always be written in a way that promotes readability." - Guido van Rossum

"It's a long-standing principle of programming style that the functional elements of a program should not be too large. If some component of a program grows beyond the stage where it's readily comprehensible, it becomes a mass of complexity which conceals errors as easily as a big city conceals fugitives. Such software will be hard to read, hard to test, and hard to debug." - Paul Graham

Why does the internal structure of a program matter?

- Programming is a collaborative activity, so it is important to know how to write code easily understandable by other humans.
- Many programmers / computer scientists / software developers / collaborators not want to continue to work with you if you routinely churn out unreadable and ugly code.
- Writing clean code reduces the chances that our code is buggy. And if there is a bug it is **much, much easier to find the bug in well written code. I see this routinely in office hours.**
- Many people in industry and in open source projects will conduct code reviews on your code and will not allow changes that do not match their internal standards. **The way we grade programs for hygiene mimics this industry best practice.**

A program is ***externally correct***, if it produces the correct results for valid inputs. Many people believe that when writing code, the only thing that matters is if your code is working or not. **This is false** - while it's important to get your code working, it is just as important to make sure your code is understandable by **other people.**

Programming is a highly collaborative activity - it's fairly common to have to work together with other people, sometimes even hundreds of them at a time. As a result, it's very important to write code that is easy to work with to allow others to integrate seamlessly with your work. Another benefit is that since clean code is easier to understand, you'll work much faster and introduce fewer bugs.

All programming assignments in CS312 and most in CS314 are done individually. However, it is still important to write readable code that follows our course hygiene guidelines - it's the core difference between whether a program is pleasant or impossible to work with.

In general, focusing on writing clean code forces you to acquire an eye for detail. The same holds true when writing an English paper or a Chemistry lab report. Programming, and writing correct (externally and internally) programs requires careful attention, so it's better to acquire that habit sooner, rather then later. As a programmer you have a responsibility to write as clean and well-designed code as possible to minimize the chances of bugs and accidents that might sneak in due to carelessness. The sooner you can get into the habit of caring about detail and correctness, the better.

Note, these are our guidelines for Mike Scott's CS314 sections. They will differ from other style guides. It does not mean one is right and one is wrong. In some cases it is simply a matter of preference. That said, we expect you to follow these guidelines in CS314.

1.3 Common Issues

Although most of these are covered later, these are some of the easiest to spot items and therefore the ones that are most often lead to points lost on assignments.

1.3.1 Magic Numbers

Typically any literal int in a program has **some** meaning. While that meaning may be clear and obvious when you write the code, it will not be clear and obvious to someone else looking at your code later, including your future self.

Therefore for any literal ints besides -1, 0, 1, and 2, create and use a constant.

Example. Count the number of Strings in array of Strings that have a length greater than or equal to 5.

```
final int MIN_LENGTH = 5;
int count = 0;
for (String s : data) {
    if (s.length() >= MIN_LENGTH) {
        count++;
    }
}
```

1.3.2. Limit methods to 25 lines of code

A method that is more than 25 lines of code is likely trying to do too much or has redundant code. Break longer methods up into smaller methods to provide structure and / or remove redundancy. Better yet, try to devise a simpler solution.

1.3.3. Minimize the scope of variables to the smallest necessary.

Don't declare variables until you need them. This helps reduce the cognitive load of trying to understand code. Following this guideline solves so many other issues such as minimizing the number of parameters to a method, minimizing the number of instance variables for a class, and not using static variables as a form of global variables.

1.3.4. Use descriptive identifiers

i, j, and k are fine for loop control variables when there is no other meaning attached to the variable, but in general we want to name our classes, methods, variables, and constants based on what they store or do.

1.3.5. Comment judiciously (but do comment!)

Yes, you must comment your code, but read the page on proper commenting. Not too much, not too little, just right. Don't simply repeat what the code is doing. Provide a higher level explanation of the method and / or algorithm. If you follow the other guidelines much of the code becomes self commented and you must only explain complex or unusual aspects of your code. Of course every method shall be commented.

2. Source File Structure

2.1 Package Statements

Do not include any package statements in your submitted code.

2.2 Import Statements

Specify each class you import separately. Do not use the wildcard. So

```
import java.util.ArrayList; // yes
import java.util.HashSet;
```

not

```
import java.util.*; // no
```

Do not include any unnecessary imports. Most IDEs have a feature to assist in organizing imports and removing unnecessary ones.

Do not import any non standard Java classes. Standard Java classes typically being with `java.` or `javax.` Compile your code on a CS department lab machine to be certain you do not have any non standard Java imports.

2.3 Ordering of Class components

In general, place class constants and instance variable declarations at the top of your class.

The order of how you place methods is up to you, but it helps readability to maintain some kind of logical ordering. One common ordering is to include all constructors in the beginning, then all other public methods before private methods. Programmers also commonly choose to put related public/private pair methods together so that another viewer of the code can see the pair at the same time.

3. Formatting

Oh, the metaphorical blood that has been spilled over code formatting.

The following section describes rules about how your code should be organized visually. It should be noted that the rules listed below are not ***inarguably*** the best rules; there are plenty of reasonable alternatives out there. **It is important, however, to adhere to the relevant style of the work environment, in this case CS314.** For other readers [your TA or instructor], code that follows the style guidelines is easier to understand than code that doesn't. Being able to anticipate conventional formatting of code eliminates time wasted on deciphering the meaning of any unfamiliar structure and formatting.

3.1 Indentation

Each time a curly brace is opened, increase the indent level. When the closing curly brace occurs and ends the code block, the indent should return to the previous indent level. The indent level applies to both code and comments throughout the block defined by the curly braces.

Choose 4 spaces to be your indent level and stay consistent.

Use spaces not tabs.

3.2 Spacing

Java is not a whitespace sensitive language, but maintaining good spacing throughout your program helps your code be more readable for human eyes. Many Java programmers follow the spacing conventions shown below for their programs.

- Class headers: `public class SomeClass {`
- Method headers: `public static void someMethod(int number1, int number2) {`
- Method calls: `someMethod(4, 3);`

- Loops:
  - `for (int i = 0; i < 4; i++) {`
  - `while (someTest) {`
  - `for (String s : list) {`
- Arrays: `int[] someArray = new int[4];`
- Variable initializations: `int someNumber = 4;`
- Expressions (a space before and after **every binary operator**):
  - `int x = 4 * (3 + 2) - 1;`
  - `boolean result = test1 && test2 || (x != 3);`

A blank line before every method comment (because **every method** is commented.)

3.3 One statement per line

Each time you finish a statement with a semicolon, you shall use a new line to begin the next statement.

Note: The exception to this rule is that the `for` loop header should have its 3 parts all on one line.

3.4 Column limit and line wrapping

No line shall exceed 100 characters in length.

When code that might otherwise legally occupy a single line is divided into multiple lines, this activity is called **line-wrapping**. Use line-wrapping to break up long lines that would otherwise exceed 100 characters, as described below.

When line-wrapping, each line after the first (each continuation line) should be indented at least one level. Common choices for how much to indent include 2 levels of indentation or lining up elements from the previous line of the long line.

Bad

It's good that this line is wrapped to maintain the 100 char line limit, but the continued line should be indented at least 1 indentation level and preferably 2.

```
  public static void method1(String param1, String param2, String
param3,
  String param4, String param5) {
    ...
  }
```

Good

This method header is properly indented and uses 2 levels of indentation for the broken up line.

```
  public static void method1(String param1, String param2, String
param3,
           String param4, String param5) {
    ...
  }
```

Good

This method header is also properly indented, and lines up elements from the broken up line.

```
  public static void method1(String param1, String param2, String
param3,
                             String param4, String param5) {
    ...
  }
```

3.5 Exceptions, especially for failure to meet preconditions

Exception code should come at the top of the method. Everything related to checking for and throwing the exception (computing a value to be used in the condition, the if statement itself) counts as exception code. Any code unrelated to the exception conditions / throwing should be placed lower than exception code. This is both an attempt to fail early to avoid unnecessary computation and a readability rule.

If an exception check cannot be checked at the beginning of the method (e.g. requires significant computation / additional statements), check for it when it can be more easily detected as the method continues.

```
// BAD! This example doesn't check/throw for the exception at the very
beginning of the method.
public static int countVowels(String name) {
    int total = 0;
    if (name.length() <= 1) {
        throw new IllegalArgumentException();
    }
    ...
}
```

Do not attach else statements onto exception checks to connect regular behavior code. Following an exception check, non-exception code should not be placed in an attached else. This is another readability rule (mostly done by convention. Another reason: why indent all of the method's regular behavior code a whole level if the structure is still clear without indenting?) as well as a logic rule. Exception code is fundamentally different than regular method behavior code, and shouldn't be connected to regular behavior with one long if/else structure - doing so almost implies the if/else branches are on the same level of importance or that they are related.

```
// BAD! This example attaches an 'else' to an exception check.
// This leads to indenting all the normal behavior code,
// which makes it a little bit harder to read.
public static int countVowels(String name) {
    if (name.length() <= 1) {
        throw new IllegalArgumentException();
    } else {
        int total = 0;
        ...
    }
}
// GOOD! This one is good! No else attached with regular behavior
// and the check for exception is at the top of the method.
```

```
public static int countVowels(String name) {
    if (name.length() <= 1) {
        throw new IllegalArgumentException();
    }
    int total = 0;
    ...
}
```

Note: `else if`s that check for more exceptions are fine. Just don't use else to have a branch for the method's normal behavior code. For example the following code structure would be fine.

```
   if (exceptionCheck) {
       throw exception1
   } else if (exceptionCheck2) {
       throw exception2
   }
   // non-exception behavior code here
```

3.6 Curly braces [Preferred, but you can use alternative brace styles]

Again, the metaphorical blood that has been spilled over brace styles is astonishing. The brace rules mentioned below document the style that you will likely see your TAs and lecturers use, **but this is optional if you want to use a reasonable (readable) alternative instead. The most important thing is consistency with whatever style you choose.**

- No line break before the opening brace.
- Line break after the opening brace.
- Line break before the closing brace.
- Line break after the closing brace, *only if* that brace terminates a statement or terminates the body of a method, constructor, or class. For example, there is *no* line break after the brace if it is followed by `else`. ***Prefer the cozy else structure.***

Example:

```
public static void method1(int times) {
    for (int i = 0; i < times; i++) {
        if (i % 2 == 0) {
            System.out.print(i);
        } else {
            System.out.print("x");
        }
        System.out.println(" hello");
    }
}
```

3.7 Enclose all blocks with curly braces

Although it is not syntactically necessary to enclose single statement blocks controlled by a loop or a conditional, do so in your programming assignments. Omitting these braces can result in maintainability issues if a later editor of your code attempts to add additional lines to the block

but fails to add the braces. **(Note, some examples and exam questions will omit braces in order to conserve visual space.)**

Example:

```
// #BAD! on assignments
for (int i = 1; i <= MAX; i++)
    if (i % x == 0)
        System.out.print(i + " ");

// #GOOD!
for (int i = 1; i <= MAX; i++) {
    if (i % x == 0) {
        System.out.print(i + " ");
    }
}
```

3.8 Remove unnecessary and unneeded code

Consider the program you turn in your final draft. Like the final draft of an English paper or Chemistry lab report, include only what is necessary.

For you solution code especially, remove any and all debugging code, notes to yourself, TODO's, and previous versions of your solution, unused imports, and other comments and / or code that is no longer necessary for your solution.

You can of course save another version of your program with that debugging code and early attempts at solutions, but **the code your turn in shall be pristine and without any unnecessary cruft. Take some well-deserved pride in what you have created.**

4. Commenting

"I coded this a few days prior and have now come up with an alternate solution that makes sense to me; however, **now looking back on this previous code I wrote, it makes no sense to me...**" **- Anonymous CS314 Student, Fall 2024.**

4.1 Motivation

Comments are specially marked lines of text in code that can be used to describe what's happening in a program. Java ignores comments when it runs a program, so they're not necessary for actually making code work properly.

There's no exact formula or algorithm for writing good comments. The simplest way to know if comments are sufficient is to ask, "If I didn't understand the code in this method,

would I still be able to use it properly by reading only the comment, the method header, and the variable names?"

The purpose of comments isn't necessarily for the author's benefit, though they can be helpful as programs get more complex. Rather, comments are for the benefit of **clients** who are using the program. Ideally, a client should only have to read the method comment and the method signature (return type, name, parameters) to know how to use a method. It will shock you when you go back and look at code a few months after you wrote it. What seemed crystal clear at the time may well baffle you later. [Likely a function of our limited short term memory.] I have often thought when I look back at code I wrote, **"Did I actually write this? What was I thinking?!?! How on earth does this actually work??"**

As such, comments should describe the behavior of a method (what it does) without its implementation (how it does it). If someone is really interested in seeing exactly how a method works, they can read the code, which will be easy to read because you followed the rest of these best practices.

4.2 Format of comments and general guidelines:

- Your comments should all be formatted consistently. You can use pre/post, JavaDoc, or some other format that includes all the required information.
- Use either multiline comments (/* */) or single line comments (//).
- No line shall be longer than 100 characters wide. Having long lines decreases the readability of your code
- Each comment should have the same level of indentation as the code it is commenting

Complete the header we provide. Include it in all .java files your submit.

The top level program comment describes what the program does without including implementation details.

Method comments in particular shall be in either pre/post format, standard JavaDoc format, or some other format that includes all the required information. Use proper capitalization; do not write comments in all capital letters.

Keep comments short and sweet. Most methods can be described in a single sentence, though more complicated methods might take two or three sentences.

4.3 Write for the client

The **client** is a person who wants to use a program without knowing how it works. For examples, see the Java API documentation (e.g. ArrayList).

4.4 Avoid implementation details

Avoid things that describe how the program or method achieves its results. In particular, avoid mentioning:

- Local variable names, private field names.
- For loop, while loop, if/else, recursion, inner data structures, calls to other methods.

If some implementation detail is vitally important, reformat the detail in a client-oriented way. For example, rather than "Uses a SortedMap to maintain keys in sorted order," restate the detail as, "Maintain keys in sorted order."

4.5 Describe parameters, results, and exceptions

Describe what each parameter represents, either in a list or as part of the method description. Describe why each parameter is needed. Comment any special cases the client may care about, e.g. "The given list should be in sorted order."

If the program produces output, describe it.

Where does the output appear?
    On the console, in a file, etc.
What is the output?
    A list of numbers, a paragraph, the result of a computation, etc.
What is the output format?
    On multiple lines, in reverse order, separated by spaces, etc.

If the program returns a value, describe what the returned value represents in relation to the input. Comment on any special cases the client may care about. For example, does the method return null or -1 in particular cases?

Tell the user precisely what will cause an exception. Include the exact type of any exceptions that are thrown next to their causes so that the client knows what could happen.

4.6 Internal comments, private methods, and fields

While method comments should not include implementation details since they're intended for the client, internal comments placed within the body of a method might be helpful for maintaining the program in the future, **especially if the code is complex or achieves a result in a clever, non obvious way.**

Do not comment trivial code, only code that is important, confusing, or tricky.

Method comments for private methods follow the same rules as public methods, but are slightly less strict. It's still important to avoid commenting on implementation details.

Put a brief comment on each class constant and instance variable describing what it represents, **if the function of the field is not immediately apparent from its name**. Comments on public fields or constants in particular should avoid mentioning implementation details.

4.7 Be direct, be correct

Reword sentences to jump straight to the point. Instead of, "This method calculates a sum…" just write, "Calculates a sum…"

Don't say the method "should" do something or "will" do something.

```
// #BAD! This section will find the min value in the array named data.

// #GOOD! Find the min in the array data.
```

Do not copy from the assignment specification. The specification usually gives too much information for commenting. It is a good idea to draw only the information the client needs from the specification and put them in your own words.

Do not have incorrect or misleading comments and remove any unnecessary comments such as IDE added TODO comments. Remove any comments provided by the instructors that were instructions.

4.8 Case studies

4.8.1 Method comments

Consider this method that prints an introduction.

```
public static void intro() {
    System.out.println("This program computes the cost of postage required to
send mail via the"
    System.out.println("United States Postal Service. In addition to sender
and recipient,");
    System.out.println("the program will need each item's dimensions and
weight.");
}
```

The following comment is too brief. It doesn't answer the most basic question, "What does the method do?"

```
// Intro #BAD!
public static void intro() {
```

The following comment is much better. It concisely describes the method's behavior.

```
// #GOOD! Prints an introduction to the program to standard output.
public static void intro() {
```

While the next comment is more descriptive, it's arguably too descriptive. It mentions that it prints an introduction to the program, which describes its behavior, but also includes unnecessary details about the method's implementation details. If someone wants to know how the printing happens, they can read the code.

```
// #BAD! Uses four println statements to print four lines of introduction to
the program. Each line is
// different, so each one needs a separate println statement as I didn't want
to embed newline characters.
public static void intro() {
```

4.8.2 Parameter comments

Consider this method that prints a list of numbers up to and including the value passed in as a parameter. The following is a good comment for the method.

```
// #GOOD! Takes an int representing a maximum and prints a list of numbers to
the console up to and
// including max in the format: 1, 2, 3, ..., max
public static void printNumberList(int max) {
    for (int i = 1; i < max; i++) {
        System.out.print(i + ", ");
    }
    System.out.println(max);
}
```

That seems pretty straightforward. The method and its parameters are well-named, and the comment is concise but descriptive. Contrast the above comment with this example.

```
// #BAD! Lists the numbers using a for loop
public static void list(int n) {
```

It's very difficult to tell what this method does. It could print a list up to but not including `n`—an important difference. Or, it could print the list starting at `n` down to 1. It's not possible to tell if it prints a list.

The only way to know what this method does and how to use it is to read all of the code and figure it out. This can be very time consuming. It's also good to note that the name of a method and the names of its parameters help describe what the method does.

4.8.3 Methods with return values

Sometimes what a method returns might not be immediately obvious. A method that returns an `int` might be returning a result that it calculates, or it might print the result to the console and instead return the number of seconds the calculation took. It's important to describe what the return value of a method represents so that people know what to expect when they use it.

This first version of the method below is well commented. The comments are concise and correctly describe what the method takes as a parameter and what it returns. The code is straightforward and explains itself. Internal comments aren't necessary here.

```
// #GOOD! Takes an integer n as a parameter and returns the factorial of n
// pre: n >= 1
// post: return n!
public static int factorial(int n) {
    int result = 1;
    while (n > 1) {
        result *= n;
        n--;
    }
    return result;
}
```

The next comment isn't descriptive enough. It doesn't describe the input and output parameters. It calculates the factorial of what? What does the returned `int` represent?

```
// #BAD! Calculates the factorial
public static int factorial(int n) {
```

4.8.4 Internal comments

The next version of the comments contains too many implementation details and the internal comments actually make the code less readable and more cluttered. The "while loop" comments are completely redundant with the `while` statement declared in code. Java programmers reading the code will know that `n--` will "update n" without needing the comment to say so. The `return` statement includes the `return` keyword, so it's not necessary to include a comment that just restates the keyword.

```
// #BAD! Uses a result variable and a while loop to calculate the result of n
factorial. The while loop
// runs until the result is calculated, and a return statement is used to
return result.
public static int factorial(int n) {
    int result = 1; // #BAD! initialize result #NOT_NECESSARY
    // while loop #NOT_NECESSARY
    while (n > 1) {
        result *= n;
        n--; // update n #NOT_NECESSARY
    } // end of while loop #NOT_NECESSARY
    return result; // return #NOT_NECESSARY
}
```

Save internal comments for more complicated statements. For example if you are using an array of ints to store the frequency of chars and mapping the ASCII code or Unicode value of the char to the array index, that is worthy of an internal comment.

4.8.5 Implementation details

The method below has appropriate internal and external comments. The comments concisely describe the behavior of the method as well as the input parameters without discussing the implementation. The internal comment is there to describe the function of a particularly odd-looking piece of code. It is a good idea to leave whitespace before an internal comment so that it's easier to read.

```
// #GOOD! Takes a Scanner "input" and a PrintStream "output" as parameters.
Prints the contents of the
// input Scanner to output with trimmed whitespace and line breaks every 10
tokens. Existing line
// breaks in the input are ignored.
public static void addLineBreaks(Scanner input, PrintStream output) {
    int tokenCount = 0;
    while (input.hasNext()) {
        output.print(input.next());
        tokenCount++;

        // Add a line break every 10th line
        if(tokenCount % 10 == 0) {
            output.println();
        }
    }
}
```

The next method description isn't sufficient. It doesn't correctly answer the questions, "What does this code do?" and, "What are the parameters?" If someone were to read these comments, they might be confused about what the `Scanner` was supposed to contain and where the output is printed.

```
// #BAD! prints out a file with line breaks every 10 tokens
public static void addLineBreaks(Scanner input, PrintStream output) {
```

The next method comment isn't appropriate because it deals with implementation details. We don't need to know that the code uses a while loop in order to use it. If someone wants to find out how the code gets the job done, they can read the code.

```
// #BAD! Uses a while loop and a scanner to output the Scanner file
// with line breaks every 10 tokens.
public static void addLineBreaks(Scanner input, PrintStream output) {
```

4.8.6 Pre conditions and post conditions AND exceptions

Exceptions are an important part of the behavior of the method. When commenting a method that may throw an exception, it's necessary to include the type of the exception and the conditions that throw it in the comment. It's also helpful to provide a short explanation of why the exception was thrown in the `Exception` itself. This is a nice way to give a user a little more information than just the exception type.

A**ll pubic methods of a class shall have pre conditions and post conditions document in the method comment.** Pre conditions and post conditions are a way to assert what must be true before and after a method runs. For the **`findMax`** method to run properly, the pre-conditions for valid parameters must be true (or else an exception will be thrown). A post condition is used to describe the effects of the method after the method has completely executed. A method that sorts a list, for example, has the post condition that the list is sorted. Some methods may not have a post-condition or (more rarely) may not have a pre condition.

```
/*
```

```
 * #GOOD!
 * pre : start >= 0, end <= list.length, start >= end
 *       Throws an IllegalArgumentException if preconditions not met.
 *
 * post: Returns the maximum value found in the list of integers between
start and end,
 *       start inclusive and end exclusive.
 */
public static int findMax(int[] list, int start, int end) {
    if (start < 0 || end > list.length) {
        throw new IllegalArgumentException("start or end out of range");
    } else if (start >= end) {
        throw new IllegalArgumentException("start is greater than end");
    }
    int max = list[start];

    // Find the maximum in the given range.
    for (int i = start + 1; i < end; i++) {
        max = Math.max(max, list[i]);
    }
    return max;
}
```

4.8.7 Commenting class and instance variables.

Fields, representing the state of the class, should be commented as well. Describe what the field represents. These comments should not be lengthy, but they should help anyone reading your code to understand why the field is important.

However, be careful to only provide comments that add information. If the field's name is very self-explanatory, then it might not need any comment.

In the following example, the first two comments are good. They describe briefly what the field stores and the format of the data. The last comment is poor because it merely repeats the name of the field. A better comment might be, "The student's age in years."

```
public class Student {
    // #Good! The first and last name of the student.
    private String name;

    // #Good! The student's score (from 0 to 100) on their latest test.
    private int score;

    // #Bad! #NOT_NECESSARY The student's age.
    private int studenAge;
}
```

5. Class Design

5.1 Fields aka Instance Variables

5.1.1Initialize fields inside the constructor

Do not initialize fields at declaration. This is both a flexibility and readability rule. It's possible that you want to initialize a field to different values in different constructors. Initializing a field at declaration and again in a constructor is both redundant and distracting.

```
// #BAD!
private ArrayList<Object> container = new ArrayList<>();

// #GOOD!
private ArrayList<Object> container;

public SomeClass() {
    container = new ArrayList<>();
}

public SomeClass(int initialCapactity) {
    container = new ArrayList<>(initialCapactity);
}
```

5.1.2 Declare all fields private access

This modifier ensures clients of the class cannot modify the internal data without using an allowed (`public`) method.

Note

> There is an exception to this rule. For node classes (`ListNode, DoubleListNode, TreeNode, BSTNode`), it is acceptable (and encouraged) to use `public` fields so that the client (you) can directly manipulate the field references to other nodes without using setter methods.

Why? As you'll see when nested and inner classes are introduced, ideally these classes would be `private` nested or inner classes, as clients should be abstracted away from the nodes themselves. Being a private nested or inner class also means that the field access modifiers wouldn't matter. However, if we write them as separate classes, public field access is the way to allow the nodes to be manipulated in the natural/conventional way.

5.1.3 Do not have unnecessary fields

The more fields your class has, the more difficult it becomes to maintain and reason about your code. With more fields, each instance of the class will also take up more memory. This is a case of the overarching guidelines of **minimize the scope of variables to the smallest necessary**.

When revising your code, watch out for fields that are only used in one public method, and/or that are recomputed/cleared every call to a specific method. These can likely be simplified to local variables within the scope of that public method/its private helper methods. **Also watch out for redundant fields that can be inferred from the properties of other fields.**

5.1.4 Accessing fields

Prefer consistency when accessing fields. Assuming a getter exists for a field (which is **not** always the case) you can access the field in several ways

```
public class Point {
    private int x;

    public int getX() {
        return x;
    // other code not shown
}
```

We have the following options for accessing the field x in instance methods of the class:

```
public void accessXExamples() {
    int temp1 = x; // direct, this. implicit
    int temp2 = this.x; // explicit this.
    int temp3 = getX();
}
```

Assuming there are no efficiency concerns (the getter method is O(1)), pick one style and use it **consistently** throughout your code.

If you prefer the direct, implicit this. approach (as most of the instructional staff does) it is acceptable to use the explicit this. when there are multiple objects of the same type as the calling object in scope. This is not required, but it is acceptable if it helps your understanding of which objects are being dealt with.

```
// #ACCEPTABLE This is an example method from the Point class.
// pre: other != null
// post: return the distance between this and other
public double distance(Point other) {
    temp result = Math.pow(this.x - other.x, 2); // this.x could be replaced
with simply x, but this use is acceptable
    temp += Math.pow(this.y - other.y, 2);
    return Math.sqrt(temp);
}
```

5.1.5 Maintain Encapsulation.

Just as 5.1.2 requires the vast majority of fields (instance variables) to be private, it is important to maintain encapsulation. If you have a field that is a mutable object, such as an **ArrayList**, do not have a getter method providing a reference to that mutable object to a client. Doing so can lead to logic errors as the client may alter the object is some way.

If feel you **must** provide an instance variable that is a mutable object to a client, then make a copy of it to provide the client. This should likely be avoided as well though, because making the copy requires using both time and space. Ask yourself, why does the client need this data? Can I instead write a method that provides the client the answer they are looking for?

5.2 Class Variables and Constants

5.2.1 Prefer class constants

A static field belongs to the class as a whole. All objects of the class type share it as opposed to fields / instance variables. Every object of the class type has its own copy of fields / instance variables. Early programmers often misuse class variables as a form of global data, leading to code that is hard to understand, hard to debug, hard to maintain, and often leads to incorrect behavior.

While class variables can be useful in some rare occasions, prefer (in CS314) class constants, declared **`final`**.

5.2.2 Use the correct access modifier.

`public` allows a client to directly interact with the constant. `private` means the constant is only accessible inside the class itself. Decide if clients need access to the class constant or whether you are using it to improve internal readability of the class.

Create and use class constants when directed by the spec and where appropriate. A good creation/use of a constant constitutes replacing a raw value that is used multiple times in your program with a constant that stores the value, but with more descriptive and readable name. This also provides more flexibility if the raw value ever needs to change.

Note

Class constants should be initialized at declaration unlike how fields should be initialized in the constructor.

Watch out for using raw numbers aka "magic values" in your code. These values might confuse another reader of your code, so you should avoid them and consider other options. There may be a useful method or way to access the value that is more logical, or the spec might have more information. **You must determine if a constant should be local to a method only or a class constant available to the whole class. If the constant is used in only one method it should be local to the method. If it is used in 2 or more methods prefer making it a class constant.**

5.3 Methods

5.3.1 Access modifiers

Methods (and constructors) to implement that are listed in the spec should all be declared `public`, so that clients can call these methods. **Any additional methods/constructors you write in the class should be declared `private`,** so that these methods can only be called internally. This will not only prevent clients from calling these methods in ways that might corrupt your

fields or otherwise break your program, but also simplify the functionality a client will have to deal with.

5.3.2 Parameters

Avoid unnecessary parameters. For the same reasons to cut down on fields, simplifying the number of parameters can simplify the code. Watch out for unused parameters or parameters whose values can be inferred by other parameters. Likewise, don't declare a variable until it is needed.

5.3.3 Returns

Avoid unnecessary returns. If you pass in a reference to an object as a parameter, there's no reason to define your method to return that same reference. Wherever called the method already has a reference to the object, as it was able to pass it in the first place, so returning the same value back is redundant. In fact, defining a method to have a redundant return may imply that you made a copy of the object passed in, and may confuse a reader of your code.

Bad

This method unnecessarily defines a return that always returns the reference to the original object. Because whoever called this method has a reference to the `List` object, we can change this method's return type to void to simplify the method.

```
public List<String> generateNames(List<String> toFill) {
    // add some elements to toFill
    ...
    return toFill;
}
```

Good

```
public void generateNames(List<String> toFill) {
    // add some elements to toFill
    ...
}
```

Do not use returns for empty statements in void methods.

```
// #BAD!
public void fooBAD(int x, int y) {
    if (y == 0) {
        return; // avoid
    } else {
        System.out.print(x / y);
    }
}

// #ACCEPTABLE!
public void fooGOOD(int x, int y) {
    if (y != 0) {
        System.out.print(x / y);
    }
}
```

Prefer to have a single return for non-void methods. However, it is acceptable to use multiple returns if the code is succinct and it is easier to simply return when we have found our answer. Likewise, in recursive methods multiple returns can lead to easier to understand code when we have a return for base case(s) and recursive cases, successful and failures.

```
// #ACCEPTABLE!
// pre: data != null
// post: return the first index of tgt in data or -1 if tgt is not present
public void indexOf(int[] data, int tgt) {
    for (int i = 0; i < data.length; i++) {
        if (data[i] == tgt) {
            return i;
        }
    }

    // never found tgt in data, never returned, tgt not present in data
    return -1;
}
```

5.3.4 Unspecified behavior

Do not add additional behavior beyond what's described in the spec. This includes additional exceptions. Only include exceptions listed in the specification for each method.

Do not modify object parameters passed to public methods, unless allowed by the spec. An unsuspecting client of the method may not have wanted their data dismantled while your method ran for a different purpose.

5.3.5 Methods do one thing

Methods should do one thing. A method should have one job. Methods that try to do to many things shall be broken up into smaller methods to provide structure.

5.4 Constructors

- When applicable, **reduce redundancy by using the `this()` keyword to call another constructor** in the same class. Generally, the constructors that assume more defaults (fewer parameters) call the more general constructors (more parameters).
- Typically, clients of a class should not have to set fields of an object immediately after construction. There should be a constructor included for this situation. The general behavior of a constructor is to set up the state of an object correctly, and this should encompass setting most fields.
- Clients of a class should never have to pass in dummy values to a constructor to set up the desired state. Instead, there should be a constructor included so that only the relevant parameters need to be passed.
- Only implement constructors that are used somewhere.
- It is acceptable to add private constructors not required by the specification if it helps reduce or remove redundant or inefficient code.

5.5 Scope

Minimize the scope of variables to the smallest necessary. Remember that the scope of a variable is the closest pair of curly braces that surround it. In the following example, the variable `nextScore` has been declared in the scope of the entire method, but note that declaring it in such a wide scope is actually not necessary.

Bad (some older programming languages often required local variables to be declared at the start of the function. This is no longer true in most modern programming languages.)

```
public static void printTenScores(Scanner console) {
    int nextScore;
    for (int i = 1; i <= 10; i++) {
        System.out.print("next score? ");
        nextScore = console.nextInt();
        System.out.println("score " + i + " = " + nextScore);
    }
}
```

Notice that `nextScore` is only ever accessed inside of the for loop, so we can actually localize its scope to just the loop itself rather than the whole method.

Good This does not cause an efficiency issues. The goal is to reduce the cognitive load of someone trying to read and understand our code. If they are faced with 10 local variables at the start of a method, it can be much harder to understand the method. Reduce the cognitive load by not declaring a variable until it is necessary.

```
public static void printTenScores(Scanner console) {
    for (int i = 1; i <= 10; i++) {
        System.out.print("next score? ");
        int nextScore = console.nextInt();
        System.out.println("score " + i + " = " + nextScore);
    }
}
```

By localizing scope, our code becomes simpler to reason about, because we've minimized the number of different variables floating around at any given location in our code. Note that eliminating unnecessary fields is often just an example of localizing variables. Also note that "re-declaring" the variable `nextScore` inside the loop is just fine efficiency-wise because of how the compiler will optimize our code. In this course, you shouldn't ever have to worry about micro-optimizations like the number of variable declarations.

5.7 Preconditions and Exceptions

Check method preconditions if possible and throw an appropriate exception if the precondition is not met.

Ensure the messages included with exceptions are meaningful. An exception without a message describing the failure provides little value over simply terminating the program, while one with a message describing the reason the exception was thrown can save a substantial amount of debugging time.

```
//#BAD!
// pre: data.length >= 7
public void printMovingAverage(int[] data) {
    if (data.length < 7) {
        throw new IllegalArgumentException(); // Not enough information.

//#GOOD!
// pre: data.length >= 7
public void printMovingAverage(int[] data) {
    if (data.length < 7) {
        throw new IllegalArgumentException("The length of the array data must
be >= 7. "
                "length of parameter data = " + data.length);
```

5.8 Remove unnecessary and unneeded code - REPEATED

Consider the program you turn in your final draft. Like the final draft of an English paper or Chemistry lab report, include only what is necessary.

For you solution code especially, remove any and all debugging code, notes to yourself, TODO's, and previous versions of your solution, unused imports, and other comments and / or code that is no longer necessary for your solution.

You can of course save another version of your program with that debugging code and early attempts at solutions, but **the code your turn in shall be pristine and without any unnecessary cruft. Take some well-deserved pride in what you have created.**

6. Naming

The names of the public methods and the class must appear exactly how they appear in the spec (case matters). If these names are not exactly correct, the provided main program should fail to compile if you try to.

6.1 Casing

- Fields, local variables, and methods should be **`camelCased`**, where every word after the first begins with capitalization.
- Class names and constructors should be **`UpperCamelCased`**. This is camel casing but the first word also begins with capitalization.
- All constants, (class, local, final fields) should be **`SCREAMING_CASE`**, where words are all uppercase and separated by underscores.

Private helper methods that aide a constructor are still methods (have explicit return types and aren't used with `new`) and shall be camelCased unlike the constructor itself.

6.2 Descriptiveness

Use descriptive names that will help out another reader of your code. Giving some kind of context for what the value(s) is or giving a high-level 1 to 2 word description is usually appropriate.

For the following example, assume we're trying to name a `List` that will keep track of the names of each chapter in a book. Since we have some context for what we want to use this variable for, we can try and come up with a descriptive name.

```
List<String> l = new ArrayList<>();
```

`l` is not descriptive. It really doesn't say anything about what the values in `l` or `l` itself represent.

```
List<String> listOfStrings = new ArrayList<>();
```

Restating the type may seem better, but it doesn't actually help a reader understand the code any more easily. A reader still doesn't have any context or high level understanding what `listOfStrings` represents.

```
List<String> values = new ArrayList<>();
```

Every variable stores a value, so this one is a little bit too vague to be helpful.

```
List<String> chapterTitles = new ArrayList<>();
```

Good! This one gives some context for what the list represents.

```
List<String> chapterTitlesForTheCurrentBookObject = new ArrayList<>();
```

This is too much. It is possible to make variable names too long. To include unnecessary information.

While one-letter names may be easy to type, they aren't descriptive. Other readers of your code will likely ponder over their meaning if they need to understand what your code is doing.

There are some appropriate uses, however, like for `for` loop variable names. Here it is acceptable to use i, j, k, etc. if the loop variable has no meaning beyond a counter. Likewise you

can use r and c (for row and column) when iterating through a 2d array or i and j (as typically used in mathematics).

Likewise, for a Point object x and y are perfectly variable names for the Point's x and y coordinates.

They're also used to describe values that are temporary or so general that there isn't really any context to describe. But for the most part, try to use more helpful names when you can think of them. Other readers will likely appreciate it.

However, if a loop control variable has some meaning beyond a simple counter than it is acceptable and desirable to give it a more meaningful name.

```
// print out a table of equivalent degrees Celsius for degrees Fahrenheit
from 0 to 100.
final int MAX_TEMP = 100;
final int FREEZE_POINT_F = 32;
final double CONVERSION_FACTOR = 5.0 / 9.0;

for (int degreesF = 1; degreesF <= MAX_TEMP; degreesF++) {
    int degreesC = (int) ((degreesF - FREEZE_POINT_F) * CONVERSION_FACTOR);
        System.out.println(degreesF + " degrees Fahrenheit is roughly " +
degreesC
                + " degrees Celsius.");
}
```

6.3 Method names

Method names are typically verbs or verb phrases as the method does something. It accomplished some task. Note, this is not always the case, but prefer method names that describe what the method does (but not how it does it.).

7. Data Types

7.1 Type parameters (generics)

Always include the type parameter when dealing with generic classes such as `ArrayList`, `TreeSet`, etc. Failing to do so will likely lead to casting that clutter up the code and/or unsafe type operations.

Bad

```
Stack visited = new Stack();
```

Good

```
Stack<String> visited = new Stack<>();
```

7. 2 Casting

Avoid casting of object types. Casting to object types is essentially disregarding the safety net of the type system that Java provides. It may achieve the desired behavior, but it is usually a sign that we should change the logic of our program elsewhere. Casting to object types should only be done if truly necessary, typically in an equals method and sometimes in order to write a more efficient version of an algorithm.

Casting to primitive types to actually convert values is fine, however, e.g.

```
int truncated = (int) (Math.random() * 5);
```

The place you will most likely cast objects is when overriding the equals method. We do not want to overload equals so the parameter shall be of type Object. Use **`instanceof`** operator or the **`getClass()`** method, to check the parameter is the same type as the **`this`** object before casting.
If the class uses generic types, **do not attempt to cast to the generic type parameter. Use the wildcard or Object as the data type of the elements of the target of the cast.**

Example:

```
public class GenericList<E> {
    private E[] container;

    public boolean equals(Object obj) { // NOT GenericList<E> obj or
GenericList obj. That overloads equals
        if (obj instanceof GenericList<?>) { // obj instanceof GenericList<E>
is a syntax error

            GenericList<?> otherList = (GenericList<?>) obj;

            // GenericList<E> otherList = (GenericList<E>) obj is NOT SAFE
and not allowed in 314.
```

7.3 Prefer simple data types

Sometimes we have options for how we want to represent the data and either option(s) will be able to produce correct behavior. Instead of using a more complicated representation that can do everything we want and more, we should try use the representation that fits the specific problem the best. The intent and logic of your code will be simpler to another reader. Some examples are listed below.

- If you're using an `int` to keep track of whether it's one value or another with only 2 choices, use a `boolean` instead.
- If you're using a `String` to keep track of a single letter, use a `char`.
- Do not use boxed types unless you need them to store in a generic data structure. If you're using `Integer` because you want to keep track of a sum, just use `int`. Likewise for `Character` and `char` AND `Double` and `double`.

The point of a data structure is to store related values. If a data structure is only used to store one or two values, we prefer to use those value(s) directly.

8. Efficiency and Redundancy

A lot of what we define to be important for efficiency boils down to “don’t do anything unnecessary”. Knowing what’s necessary and what isn’t can be difficult, but following the suggestions of the assignment specifications for certain implementation details will lead you in the right direction. Unfortunately, there’s no hard and fast rule for avoiding every inefficiency. To avoid them, you will have to dedicate some time to reasoning about your code. In other words you may find it useful to perform algorithm analysis as taught the first week of class on the code you write.

As a general rule, reduce or remove complicated expressions and logic where reasonably possible.

8.1 Creating new objects

Avoid making objects that you don’t need. This can be tricky to spot, but in particular you should review your code statements that instantiate `new` objects or call other methods that do so. Making sure that every object you instantiate is necessary (no other available methods can provide the same or reasonable behavior) and logical will give some sense of security here.

The first example makes a `new ArrayList<>();` just to discard it in the next line. When we set `words = generateWords()` in the second line, the previous value of `words` is lost (unless it’s saved elsewhere). The second example doesn’t throw away any newly made objects.

Bad

```
ArrayList<String> words = new ArrayList<>();
words = generateWords();
```

Good

```
ArrayList<String> words = generateWords();
```

Likewise, avoid building up a String inside a loop. Strings are immutable. A program that builds up a String inside of loop using String concatenation, creates a new String with each concatenation. Also, building up a String in a loop can be inefficient. **Prefer using a StringBuilder when building up a String via a loop.**

Bad

```
// assume data is an array of Objects
String result = "";
for (Object o : data) {
    result += o + " ";
}
return result;
```

Good

```
// assume data is an array of Objects
StringBuilder sb = new StringBuilder();
for (Object o : data) {
    sb.append(o);
    sb.append(" ");
```

```
}
return sb.toString();
```

8.2 Recomputing values

Avoid recomputing complex expressions and method calls. **If you have to use the value of a method call or a complicated expression more than once, you should store it in a well-named variable.** Using the value stored in this variable will give you instant access instead of recomputing the value by evaluating the method call or expression again. This will also clear up the clutter of your code by replacing generic symbols and expressions with a name to describe some context.

There are some methods that are fast enough that storing the value in a variable for the future doesn't save us anything. For example, the `size()` method for any data structure we use in this course will have the same instant access whether we store it or call the method. In such a case, efficiency is not an issue we have to worry about.

8.3 Factoring

If you have lines of repeated or very similar code that need to be executed in different places, group the code of the task into a private helper method. Putting the code into one place instead of several will make making changes easier, as well as make our code more readable.

Code inside if/else structure should represent different cases where the code is inherently different. This means duplicate lines of code or logic in an if/else structure should be factored out to come before or after, so that they are only written to happen once, unconditionally.

Bad

Note that there are repeated lines of logic that actually always happen, instead of conditionally like how our structure is set up. We can factor these out to simplify and clean our code.

```
if (x % 2 == 0) {
    System.out.println("Hello!");
    System.out.println("I love even numbers too.");
    System.out.println("See you later!");
} else {
    System.out.println("Hello!");
    System.out.println("I don't like even numbers either.");
    System.out.println("See you later!");
}
```

Good

```
System.out.println("Hello!");
if (x % 2 == 0) {
    System.out.println("I love even numbers too.");
}  else {
    System.out.println("I don't like even numbers either.");
}
System.out.println("See you later!");
```

Review any if/else structures you write and make sure there are no duplicate lines of code or logic at the beginning or the end.

8.4 Redundant conditional logic

This section details issues with writing unnecessary if/else logic that can be omitted. We will refer to any control flow like if, else, etc. as conditional logic.

8.4.1 Loops

Good bounds and loop conditions will already deal with certain edge case behavior. Avoid writing additional conditional logic that loop bounds already generalize.

Bad

The `if` here is redundant. The loop won't produce any output (or even run) if n were 0 or negative in the first place, so we should remove the check.

```
if (n > 0) {
    for (int i = 0; i < n; i++) {
        System.out.println("I love loops <3!");
    }
}
```

8.4.2 Already known values

By using control flow (if/else/return) properly, you can guarantee some implicit facts about values and state without explicitly checking for them. The examples listed below add in redundant conditional logic that should be removed.

Bad

Because **`someTest`** was tested for alone in the first if, future branches of the if/else structure know **`someTest`** must be false, so the check for **`!someTest`** is redundant and can be omitted.

```
if (someTest) {
    ...
} else if (!someTest ...) {
    ...
}
```

Bad

At the second `if` statement, we know that `someTest` must have been false, so it's not necessary to check its value again.

```
if (someTest) {
    throw exception/return
}
if (!someTest) {
    ...
}
```

8.5 Reuse existing code and methods

Avoid re-implementing the functionality of already existing methods including commonly used methods from the Java standard library, by just calling those existing methods.

In the following example, the two methods are almost identical, except `indexOf` is more powerful/flexible than **`contains`**. So, **`contains`** can actually just use a call to **`indexOf`** to reuse already written behavior.

```
public int indexOf(int value) {
    for (int i = 0; i < size; i++) {
        if (elementData[i] == value) {
            return i;
        }
    }
    return -1;
}
```

Bad

```
    public boolean contains(int value) {
        for (int i = 0; i < size; i++) {
            if (elementData[i] == value) {
                return true;
            }
        }
        return false;
    }
```

Good

```
    public boolean contains(int value) {
        return indexOf(value) >= 0;
    }
```

The classes you will use in this course (`String`, `ArrayList`, `TreeMap`, etc.) will provide many useful methods and you should aim to take advantage of them by using them. Rewriting existing behavior is less flexible, redundant, and by writing more lines of code you raise the risk of encountering more bugs.

This includes repeated code in the methods you write. If you have the exact same block of code, three or more lines long, repeated in a method or multiple methods, factor it out into a separate method. This includes constructors with repeated code.

8.6 Prefer code that can scale as the problem changes or grows.

Example: Assume we want to add up the lengths of four consecutive Strings in an array of Strings given a starting index.

Bad

```
    // data is an array of Strings.
    // We know no elements of data are null
    // We know index is inbounds and that the next three indices are also
    in bounds.
```

```
int sum = data[index].length() + data[index + 1].length() + data[index
+ 2].length + data[index + 3].length();
```

Good

```
int sum = 0;
final int LAST_INDEX = index + 3; // 3 can be changed to scale the
problem.
for (int i = index; i <= LAST_INDEX; i++) {
    sum += data[i].length();
}
```

8.7 Stop computing when the answer is known.

When looking for an answer or solution, stop computing when that answer is known.

Example: Determine if an array of ints contains a target value a given number of times **or more.**

Bad

```
public static boolean containsAtLeast(int[] data, int tgt, int minNum)
{
    // Always looks at every element in the array.
    int count = 0;
    for (int element : data) {
        if (element == tgt) {
            count++;
        }
    }
    return count >= minNum;
}
```

Good

```
public static boolean containsAtLeast(int[] data, int tgt, int minNum)
{
    // If we find the minimum number required, stop looking. We have
our answer.
    int count = 0;
    for (int element : data) {
        if (element == tgt) {
            count++;
            if (count == minNum) {
                return true;
            }
        }
    }
    // Did not find minNum occurrences of tgt.
    return false;
}
```

8.8 Implement compareTo and compare simply if possible

When implementing the Comparable interface we must implement the method compareTo. compareTo returns an int. The return value shall be < 0 if the calling object (this) is "less than" the explicit parameter, = 0 if the objects are "equal", and an int > 0 if the calling object is greater

than the explicit parameter. While some implementations of compareTo return -1, 0, or 1 **this is not required and can lead to unnecessarily verbose code.**

Example: We have Person class. Each person has a height in inches. The Person class implements the Comparable interface and we want to order Person objects based on the field that represents the persons height.

```
public class Person implements Comparable<Person> {

    private int heightInInches;

    // other fields not shown
```

Bad

```
    public int compareTo(Person other) {
        if (heightInInches < other.heightInInches) {
            return -1;
        } else if (eightInInches == other.heightInInches) {
            return 0;
        } else {
            return 1;
        }
    }
```

Good

```
    public int compareTo(Person other) {
        return heightInInches - other.heightInInches;
    }
```

9 Zen

9.1 Boolean Zen

Boolean Zen is all about using boolean values efficiently and concisely.

The following code shows a tempting way to write an if statement based on boolean value.

Bad

```
    if (test == true) {
        // do some work
    }
```

Note that `test` itself is a `boolean` value. When it is `true`, we are asking whether `true == true`. `true == true` will evaluate to `true`, but remember that the `if` branch will execute as long as what is in its parentheses evaluates to true. So we can actually use `test` directly:

Good

```
    if (test) {
        // do some work
    }
```

Note

To check for the opposite of **`test`**, don't check for **`test == false`**. Instead, use **`!test`** to check that the opposite of **`test`** evaluates to **`true`**.

Here's an example that uses what we learned in the previous section about simplifying boolean expressions.

Bad

```
if (test) {
    return true;
} else {
    return false;
}
```

There is actually a much more concise way to do this. If we want to return true when test is true and false when test is false, then we actually just want to return the value of **`test`**.

Good

```
return test;
```

In general, make your use of booleans to simplify conditional and boolean expressions. This is something to look out for whenever you have conditions or boolean values.

Bad

Because this code always wants to do one or the other, (and doesn't involve a return or exception) we want to express this code more simply as an if/else.

```
if (someTest) {
    System.out.println("hello!");
}
if (!someTest) {
    System.out.println("I'm redundant");
}
```

Bad

Note that the behavior inside this if block is exactly the same behavior as in the other if block. Instead of rewriting the same code twice, we can combine the two if conditions with `||` and just write the behavior once.

```
if (max < result) {
    return max;
}
if (max == 0) {
    return max;
}
```

Bad

It doesn't matter if you think of conditions/cases or their negated versions, but after revising your code don't include empty condition blocks with no line of code inside. Instead, just flip the condition to have if (max >= 0), and no else.

```
if (max < 0) {
    // do nothing
} else {
    ...
}
```

Don't use if / else statements or ternary operators to to assign boolean values the result of a boolean expression.

Bad

```
boolean meetsReq;
if (score > = MIN_SCORE) {
    meetsReq = true;
} else {
    meetsReq = false;
}
```

Still Bad

```
boolean meetsReq = false;
if (score > = MIN_SCORE) {
    meetsReq = true;
}
```

Good

```
boolean meetsReq = score >= MIN_SCORE;
```

9.2 Loop Zen

9.2.1 Loop Bounds

When writing loops, choose loop bounds or loop conditions that help generalize code the best. For example, the code before this `for` loop is unnecessary.

Bad

```
System.out.print("\*");
for (int i = 0; i < 4; i++) {
    System.out.print("\*");
}
```

Instead, why not just run the loop one extra time? This way we avoid writing duplicate behavior.

Good

```
for (int i = 0; i < 5; i++) {
    System.out.print("\*");
}
```

9.2.2 Only Repeated Tasks

If you have something that only happens once (maybe at the end of a bunch of repeated tasks), then don’t put code for it inside of your loop.

Bad

```
for (int i = 0; i < 4; i++) {
    if (i != 3) {
        System.out.println("working hard!");
    } else {
```

```
            System.out.println("hardly working!");
        }
    }
```

Good

```
    for (int i = 0; i < 3; i++) {
        System.out.println("working hard!");
    }
    System.out.println("hardly working!");
```

Similarly, a loop that **always** runs one time is also a bad use of a loop. When reviewing the logic of your code, it might be helpful to check if a loop can be reduced to an if or just deleting the loop entirely.

9.3 Recursion Zen

Avoid adding extra if/else cases to recursive methods when there are more generalized conditions to use.

Bad

This private method is an example of **arm's length recursion**: it has an extra base case for `index == list.length - 1`. If we omit this and just use the first base case, the behavior will actually still generalize and be correct, but our code will be more concise and easier to reason about.

```
private static int sum(int[] list, int index) {
    if (index == list.length) {
        return 0;
    } else if (index == list.length - 1) {
        return list[index];
    } else {
        return list[index] + sum(list, index + 1);
    }
}
```

Good

```
private static int sum(int[] list, int index) {
    if (index == list.length) {
        return 0;
    } else {
        return list[index] + sum(list, index + 1);
    }
}
```

Bad

This example also exhibits arm's length recursion since the code in the public method would be more appropriate to handle in the private helper method.

```
public static int sum(int[] list) {
    if (list.length < 1) {
        return 0;
    } else {
        return list[0] + sum(list, 1);
    }
}

private static int sum(int[] list, int index) {
```

```
        if (index == list.length) {
            return 0;
        } else {
            return list[index] + sum(list, index + 1);
        }
    }
```

Good

```
    public static int sum(int[] list) {
        return sum(list, 0);
    }

    private static int sum(int[] list, int index) {
        if (index == list.length) {
            return 0;
        } else {
            return list[index] + sum(list, index + 1);
        }
    }
```

10. Disallowed features

We have very specific goals in 314 concerning what we want to learn and evaluate you on. Therefore, some Java features are disallowed.

In general if a programming assignment specification does not disallow a feature, method, or class from the Java standard library and it isn't listed here, you **may use it. If you are unsure post to the class discussion group for clarification.**

10.1 Avoid the use of break and continue, especially on assignments

The keywords break and continue can be used to write cleaner code, but the vast, vast majority of beginning programmers misuse them to create code that is more difficult to understand without improving readability or efficiency. Instead of having all the exit conditions at the top of a loop the break and continue keywords are used to scatter the exit conditions throughout the code, making it harder to understand.

Therefore break and continue shall not be used on programming assignments in 314. After CS314, go nuts! Use them all you want.

You **may** return from inside of loops if it leads to cleaner and more logical code, but avoid returns buried deep inside loops.

10.2 Do not return from void method

Do not use the return statement to exit early from a void method,

10.3 Do not use Java 8+ functional features.

Those features are for another course, not the current version of CS314. We will discuss them near the end of the course if time allows.

Do not use the Java 8 functional features such as lambdas, streams, and method references.

10.4 Do not use Java 11+ local variable type inference

Do not use the var keyword or local variable type inference.

10.5 Do not use try / catch blocks to deal with avoidable logic errors.

try / catch blocks are meant for exceptional situations that cannot be handled with standard control structures. For example, if we are reading from a data source (file) on a remote computer and that data source is deleted while our program is running, we cannot avoid that with standard control logic. Thus the use of a try catch block for an exceptional situation outside of our control.

Compare that to looping through an array. Some programmers misuse try / catch blocks instead of standard control structures. This makes the code confusing as we expect try catch blocks to be used for those exceptional circumstances.

Example. Given an array of ints add up the first 10 elements of the array or all the elements of the array if the length of the array is less than 10.

Bad

```
int sum = 0;
try {
    for (int i = 0; i < 10; i++) {
        sum += data[i];
    }
} catch (ArrayIndexOutOfBoundsException aiobe) {
    // oh well, less than 10 elements, nothing to see here, move along
}
```

Good

```
int sum = 0;
int i = 0;
int LIMIT = 10;
while (i < data.length && i < LIMIT) {
    sum += data[i];
    i++;
}
```

10.6 Do not use any classes from the java.util.concurrent package.