Java Coding Standards

Development:

For code development, we should use the following these steps strictly in this order. Make sure to complete each step fully before continuing with the next step.

1. Implement the skeleton of the class structure/relationships, including all methods, major attributes, etc. The methods should already have JavaDoc, The attributes should already have comments, etc. Do not add any implementation to the methods. The code should compile. (e.g., if a method returns an object, simply return null to make the code compile).

2. Inside the various unimplemented methods, add comments for the implementation you plan to do.

3. Fill in the implementation between your comments.

4. Run tests and fix broken code.

Commenting:

1. Add the following to the beginning of all of your source files:

```
/***
*
*  Class: <CSI class name> <What is the class about>
*
***/
```

2. Add JavaDoc to all of the code, including classes, methods, etc. Before submission make sure that you can run the
JavaDoc export without any errors or warnings.

```
/**
* Adds two ints (a+b) and returns the result *
* @param a first integer to add
* @param b second integer to add *
* @returns the sum of a+b
* @throws OverflowException
* if a+b exceeds the value representable by int */
public int add(final int a, final int b) {
```

Note that JavaDoc follows a specific form:
- There are no dashes (-) between parameter and definition
- Use @ to define values like @author, @param, etc.
- Do not add extraneous information to method JavaDoc like method name.

3. Individually and meaningfully comment member variables and class constants.

4. Obvious/obfuscated comments are useless. Do not use them.

5. Properly (but reasonably) comment the code. A developer should be able to get a general idea of what's going on by just reading comments (and no code).

6. Each element needs a definition. This includes @param
@throws SpecialException // Bad!
@throws SpecialException if val is null or fails validation // Good!

7. Check your comments for spelling and grammatical errors.

Coding:

1. You may only use the standard Java library and JUnit unless otherwise notified. Do not include any additional JARs in your build path (and do not allow your IDE to automatically do so).

2. Follow the Java Coding Conventions from inventors of Java (see class page for document).

3. Do not use tabs. Always use braces for code blocks, even for a single line of code. For example,

```
if (x == 5) { System.out.println("True!");
}
```

The same rule applies for while, for, etc.

5. Import only necessary classes. Do not use wildcard imports (e.g., java.util.*) unless there are 4 or more classes from that package.

6. Add any and all annotation hints to your code (e.g., @Override).

7. Eliminate code replication.

8. No spurious object creation.

```
String firstName = "";
String lastName = new String("");
// Assignments above wasted assignment/allocation since just replacing values firstName =
in.nextLine();
lastName = in.nextLine();
```

9. Do not call toString() if it is implicitly called.

10. Do not use deprecated methods.

11. Avoid resource leaks. Examples include
    a. Failure to close files
    b. Memory leaks

12. Always specify access (or comment why package is appropriate). Use correct access.

13. Use foreach variant of for-loop if applicable.

14. Use collection interface references instead of concrete type references. Also, make sure to use the diamond operator.
ArrayList l = new ArrayList(); // NO! List<String> l = new ArrayList<>();
15. Don't use the older collection classes such as Vector and Hashtable. Instead use ArrayList and HashMap. The main difference is that the new collection classes are not synchronized so their performance should be better.

16. Catch the most specific exception type.

17. Move all literal constants to variable constants except in really obvious situations.

18. Make error messages as useful as possible. ("Parameters bad" vs "Usage: go <file> <date>").

19. Declare variables with use in the minimum scope.

20. Do not use global variables unless absolute necessary. Make sure the explanation for needing global variables is clearly commented. Global constants are fine.

21. Avoid if-else if- else if there are too many conditions. Instead use switch-case.

22. A switch statement should always have a default.

23. Insure the flow of the code is easily understandable.

24. Insure variable and method names meaningful.

JUnit:

1.Provide useful test names (e.g., testTruncatedDecode() is a better name than testThing()).

2. Each test should be independent of other tests. You may not assume any test execution order.

3. Use specific asserts (assertTrue vs. assertEquals).

4. Keep your tests small by limiting the number of failures a test reports. A test failure should indicate one particular problem. Consider refactoring long tests.

5. Properly test exceptions according to your JUnit framework best practices.

6. Don't just test the happy path. Include boundary conditions, etc.

7. JUnit tests should print nothing.

8. Test the coverage of your code by your tests.