Java Coding Guidelines

All code must follow best practices. Part (but not all) of this is adhering to the following guidelines:

Development

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

1. Design your approach on paper, including class structure/relationships.
2. Implement the skeleton of your class structure/relationships, including all methods, major attributes, etc. Your methods should already have JavaDoc, your attributes should already have comments, etc. Do not add any implementation to the methods. Your code should compile. (e.g., if a method returns an object, simply return null to make the code compile).
3. Fully implement testing based on the specification. For each element of the specification (e.g., throws exception if name is null), write a test. Your test should be complete and compilable/executable. Of course, they will mostly fail because your classes have no implementation.
4. Inside the various unimplemented methods, add comments for the implementation you plan to do.
5. Fill in the implementation between your comments.
6. Run tests and fix broken code.

Commenting

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

```
/*********************************************
 * Author:  <Your name>
 * Assignment: <Assignment name>
 * Class:   <CSI class name>
*********************************************/
```

   To be clear, this is not the JavaDoc for the class.

2. Add JavaDoc to all of your 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 your 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.
4. 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. Eliminate code replication.
9. Seriously, eliminate code replication.
10. Properly address all compiler warnings. Do not suppress compiler warnings unless **well** justified. Include your justification in a comment.
11. 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();
    ```
12. For simple boolean methods, return directly from expression instead of using if.
    ```
    boolean empty() { // Yuck
        if (length == 0) {
            return true;
        }else {
            return false;
        }
    }

    boolean empty() { // Yep
        return (length == 0);
    }
    ```
    This avoids potential errors such as getting true/false returns backwards.
13. Do not use C-style array declarations.

    ```
    int x[]; // No!!!
    int[] x; // Yep
    ```
14. Do not call toString() if it is implicitly called.

    ```
    System.out.println(blah.toString()); // NO!!!!
    System.out.println(blah); // Yep
    ```

15. Do not use deprecated methods.
16. Use `"Mom".equals(s)` instead of `s.equals("Mom")` to handle the case of s == null.
17. Avoid resource leaks. Examples include
    - Failure to close files
    - Memory leaks
18. Always specify access (or comment why package is appropriate). Use correct access.
19. No inappropriate member variables. Member variables are for state related to object, not for variables used by several methods.

20. Don't include extraneous, non-executed, or always-executed code. Examples include
    - Constructors that would be autogenerated
    - String blah = thing; return blah;
    - if (done == true) – use if (done)
    - Assignments, comparisons, etc. with no side-effect
    - Dead code
    - Unnecessary type operations, elses, etc.
    - Unused variables
21. Use foreach variant of for-loop if applicable.
22. 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<>(); // Yep!
    ```

23. 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. Vector and ArrayList differ slightly in their expansion algorithms. Unlike Hashtable, HashMap permits null values and a null key. If you need synchronization, use the Collections class synchronization wrapper.

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

    Note that Java concurrent package contains ConcurrentHashMap and CopyOnWriteArrayList for better performing synchronized maps and lists.
24. Catch the most specific exception type.
25. Move all literal constants to variable constants except in obvious situations.

    ```
    if (size > 255) // Wrong
    if (size > MAXSIZE) // Great!
    ```

26. Do not import or specify default packages (e.g., use String not java.lang.String).
27. Do not explicitly extend Object.
28. Do not violate encapsulation.
29. Code should only print to console when appropriate. Inside a library is not an appropriate place to print to the console. Use logger if need to output in such cases. If you are printing to the console, print to the correct stream (stdout vs. stderr).
30. Make error messages as useful as possible. ("Parameters bad" vs "Usage: go <file> <date>").
31. Declare variables with use in the minimum scope. Do not predeclare at the function start. Predeclaration leads to overextended scope (entire function) and repeated initialization.
32. Run code and runtime analysis tools to identify errors.
33. Make sure all numeric constant values in your code are justified? int[52] probably cannot. Why not 51? 53?
34. Do not use global variables unless absolute necessary. Make sure the explanation for needing global variables is clearly commented. Global constants are fine.
35. Prefer

    ```
    this.attribute = Objects.requireNonNull(attribute, "attribute cannot be null");
    ```

    to

    ```
    if (attribute == null) {
        throw new NullPointerException("attribute cannot be null")
    }
    this.attribute = attribute;
    ```

    It is less code, more readable, and offers fewer opportunities to make mistakes.

36. Avoid the *if-requires-an-else* pattern. I'm **not** saying you should never use else; just only use it when appropriate. Consider the following code:

```
if (validation test fails) {
    throw Exception
}
Real Stuff
```

    This has fewer nesting levels than requiring Real Stuff to be inside an else, reducing complexity and increasing readability.
37. A switch statement should always have a default. The only exception is if you can somehow prove that you've covered all possible cases.
38. Ensure the flow of the code is easily understandable.
39. Ensure variable and method names meaningful.
40. Ensure that you would want to be given this code for maintenance and modification.

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)

   ```
   assertTrue(5 == x);
   results in java.lang.AssertionError:
   assertEquals(5, x);
   results in java.lang.AssertionError: expected:<5> but was:<6>
   ```

   In this case, assertEquals() provides more useful information. JUnit provides a wide range of asserts (e.g., assertArrayEquals, assertNull, assertSame, etc.).
4. Keep your tests small by limiting the number of failures a test reports. A test failure should indicate one 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. This is not a definitive measure for a good test; however, it can certainly show you bad (incomplete) testing.

