Coding Guidelines

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

Development

For 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 method doc (e.g., JavaDoc for Java), 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, compilable, and executable. Of course, they will mostly fail because your classes have no implementation. This is called a "test first" approach as you write the tests before the 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 documentation for the class
2. Add documentation comments to all your code, including classes, methods, etc.
   a. Java Specifics
      i. Use JavaDoc
      ii. Before submission make sure that you can run javadoc without any errors or warnings.
      iii. All source, including tests, must have full JavaDoc.
      iv. A few notes specifically on JavaDoc:
```
/**
 * 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 (e.g., Don't add the method name in the method JavaDoc; I already know the method name)

   b. Other Language (e.g., C/C++, scripting) Specifics
      i. Use Doxygen standard (but you don't have to run Doxygen command).
      ii. If your language has header files, add function docs in the header files (not the implementation). You must, of course, comment your code.
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 requires a definition. This includes parameters
```
@throws SpecialException // Bad!
@throws SpecialException if val is null or fails validation // Good!
```
7. Fix all spelling and grammatical errors in comments, variable names, etc.

Coding

Below are generic coding guidelines. Code examples may not be in your specific language; they are there to demonstrate the principles that apply to any language.

1. Eliminate code replication.
2. Eliminate code replication.
3. Seriously, eliminate code replication.
4. Properly address all compiler warnings. Do not suppress compiler warnings unless **well** justified. Include your justification in a comment. Note well that the compiler is your friend. Look at the compiler warnings page; you may want even more warnings.
5. Ensure the flow of the code is easily understandable. While there's no hard rule, generally nesting beyond three levels within a method indicates a need for refactoring.
6. Ensure variable and method names meaningful.
7. Ensure that you would want to be given this code for maintenance and modification.
8. Do not violate encapsulation. Encapsulation is in part about forcing users of a class to use mutator methods if they want to change the state of the instance (i.e., data hiding).
9. Use proper refactoring. Break up long methods. If your code includes many levels of nesting (e.g., ifs inside ifs inside whiles), consider refactoring.
10. Do not use tabs; use spaces.
11. Import/Include only what is actually necessary/used.
12. Always specify access modifiers for classes, methods, fields, etc. (or comment why package-private is appropriate). Use the most restricted access that still allows specification implementation.
13. Use casting only when necessary. Casting is asserting that you know more about the real type than the compiler. This is dangerous. If you actually do know more, explore how to better inform the compiler so it knows what you know. To be clear, casting is sometimes necessary but should be the exception.
14. No inappropriate member variables. Member variables are for state related to object, not for variables used by several methods.
15. Do not use deprecated methods.
16. A switch statement should always have a default. The only exception is if you can somehow prove that you've covered all possible cases.
17. 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.
18. 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();
    ```
19. 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.
20. Avoid resource leaks. Ex.,
    a. Failure to close files
    b. Memory leaks

21. Don't include extraneous, non-executed, or always-executed code. Examples include
    a. Constructors that would be autogenerated
    b. `String blah = thing;`
       `return blah;`
    c. `if (done == true)` – use `if (done)`
    d. Assignments, comparisons, etc. with no side-effect
    e. Dead code
    f. Unnecessary type operations, elses, etc.
    g. Unused variables
22. Use foreach variant of for-loop if applicable/available.
23. Throw and catch the most specific exception type.
24. Move all literal constants to variable constants except in really obvious situations.
    ```
    if (size > 255) // Wrong
    if (size > MAXSIZE) // Great!
    ```
25. Code should only print to console when printing is in the specification. 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).
26. Make error messages as useful as possible (e.g., "Parameters bad" vs "Usage: go <file> <date>").
27. For printing/logging errors, always include any available details such as system-specific error messages.
28. Always include information you have in an exception. Always include a message with any known specificity. Do not throw away exception information
    ```
    catch (IOException e) { // Bad
     System.err.println("Bad IO");   // Non-specific error message
     throw new RuntimeException("Bad IO");  // Loss of cause exception
    }

    catch (IOException e) { // Good: Uses message and cause
     System.err.println("Bad IO: " + e.getMessage());
     throw new RuntimeException("Bad IO", e);
    }
    ```
29. Avoid using exceptions strictly for testing to throw another exception
    ```
    try { // Bad - Creates NullPointerException unnecessarily
     Objects.requireNonNull(s, "");
    } catch (NullPointerException e) {
     throw new IllegalArgumentException("s cannot be null");
    }
    ```

    instead use

    ```
    if (s == null) { // Good - Only one exception
     throw new IllegalArgumentException("s cannot be null");
    }
    ```
    If you are calling a method that performs some function on valid input (e.g., age = Integer.parseInt(s), read(buf), etc.), then it's great to use exceptions for testing, even if you would catch and throw a different exception type. The problem is building and catching an exception for the sole purpose of error checking when it would be simple to validate without it.
30. 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.
31. Run code and runtime analysis tools to identify errors.
32. Make sure all numeric constant values in your code are justified? int[52] probably cannot. Why not 51? 53?
33. Do not use global variables unless absolute necessary. Make sure the explanation for needing global variables is clearly commented. Global constants are fine.
34. 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.
35. Remove all commented code.

36. Unless otherwise specified, programs with command line arguments should output a proper usage message and, if applicable, an error message plus return a non-zero exit code if there are problems. A proper usage message at least contains the list of parameters. For example, if the command line parameters are server and port and the user enters “abc” for the port, the program should:
    a. Print an error message like “Bad port”
    b. Print a usage message like “Parameter(s): <server identity> <port>”
    c. Exit with a non-zero code
37. Use “fast-fail” when appropriate. For example, check input first for validation failure conditions. If parameter validation failure would terminate a method, perform validation first (before computing anything).

Java

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

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

2. Use `"Mom".equals(s)` instead of `s.equals("Mom")` to handle the case of `s == null`.
3. 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).
4. Unless otherwise specified, you may only use the following packages: standard [java.lang (autoimported; do not import), java.util, java.io, java.math, java.nio, java.net, java.text, java.time], JUnit [org.junit], and your own.
5. Follow the Java Coding Conventions from inventors of Java.
6. Kill boilerplate code. For example, 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.
7. 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!
   ```

8. 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.
9. For arrays, associate [] with the type, not the variable.
   ```
   int x[]; // No!!!
   int[] x; // Yep
   int[] z[];// No!! (equivalent to int[][] z;)
   ```
10. Do not explicitly extend Object. Do not use package name for methods, types, etc. (e.g., use List rather than java.util.List) unless absolutely necessary for disambiguation.
11. Add any and all annotation hints to your code (e.g., @Override).
12. Do not use wildcard imports (e.g., java.util.*) unless there are 4 or more classes from that package.
13. Do not import or specify default packages (e.g., use String not java.lang.String).
14. Do not use reflection to load and execute. Reflection is great, and it's good to use; however, since we are not using/requiring an automated build tool, we will skip using reflection.

C/C++

1. You may only use the standard libraries. Do not use any non-standard libraries.
2. Make sure to use the proper test for functions. Any error messages should include any system error information. Check the man page for error signaling specification. For functions that set errno (like fork()), you must call perror to print system message. For functions that return an error code (e.g., pthread_create), you need to fprintf to stderr the string from strerror(<returned number from pthread_create>).
3. The command-line usage message must include the executable name (e.g., ./command <file> <date>)

JUnit

1. Provide useful test names (e.g., testTruncatedDecode() is a better name than testThing()).
2. Your test classes must all have Test in the end of their name (e.g., IntegerTest.java).
3. Each test should be independent of other tests. You may not assume any test execution order.
4. 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.).
5. Always use the assert equals for double that includes a delta of 0.0001. If you really want them to be perfectly equal, use a delta of 0.
6. 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.
7. Properly test exceptions according to your framework's best practices.
8. Don't just test the happy path. Include boundary conditions, etc.
9. JUnit tests should print **nothing**!
10. 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.
11. Do NOT create dependencies between your tests and your implementation. Your tests and implementation should only rely on the published, public API. For example, if you had a constant, MAXZ, defined in your implementation, but this constant is NOT in the public API, you may NOT use this constant in your test.