Programming Assignment Requirements

Note the following requirements for **ALL** programming assignments throughout the **entire** semester:

General

- THE OVERRIDING RULE IS: KEEP IT SIMPLE!!!!!! In general, a simpler program works for some cases is better than a complicated program that does not work for any cases. Programs that fail to compile will receive a 0.
- Your programs must demonstrate good design with efficient implementations. You are responsible for all of the implementation, including when a generated/default/inherited implementation is provided (e.g., toString in Java).
- You must precisely follow the complete specification for any assignment; however, any given specification (interface, classes, tests, etc.) defines the **minimum**. It does NOT, in any way, prevent you from having additional classes and methods. In fact, a good design will likely have additional elements. This does not prevent you from having less restrictive access permissions. Any interface specification is just for the prototype information (e.g., class/method name, parameter/return types, static or non-static). Determination of which classes/methods should be abstract and/or overridden is yours according to good design.
- Distrust any external (i.e., from network, users, files, telepathy, etc.) input.
- Your programs should not print anything to the console unless specifically instructed to do so. In particular, unit test and library classes should print **nothing** to the console. Of course, when appropriate, you should output to a log.
- Your program should gracefully handle problems such as system call failures, bad input, etc. Graceful means that it gives a useful message and follow the protocol. If terminating, your program should return a code that is useful for debugging specific failure explanation. Your program must NEVER terminate outside of your control (e.g., termination due to exceptions, faults, etc.). This includes parameter usage. Even if the book examples terminate ungracefully (e.g., by throwing IllegalArgumentException), your program may NOT do this.
- Friendly heads-up: The programming assignments may build on one another. This means that each program may depend on one or more of its predecessors.
- Make sure you follow all Coding Conventions/Guidelines and preparation instructions from the class web page, assignment instructions, and deliverable specifications.
- Test your program thoroughly. Testing should be automated. For unit tests, use a code coverage tool to make sure you test every testable line.
- Always run your tests on your final submission. Do not fully test, then make one last change that you "know" won't break anything, and then submit.

Upload

- You will submit your code as a zip file to the submission site. Your archive should contain ONLY source code files (e.g., no compiled files). Your code should be contained in directories corresponding to their packages; **no** other directories should be used in your archive. Your program must contain all files necessary for self-contained, command-line compilation and execution except standard (e.g., libc) and provided libraries.
- Any path in your code must be relative to your submission directory structure (e.g., NOT C:\Users\Phil\log.log). You can place such files at the top-level (i.e., at the same level as the top-level package directories) as this will be the default working directory for the JVM. You should test with the command-line (e.g., javac and java) to make sure it works. Log files must always be created in the JVM working directory. Servers should always start with an empty log file (no append).
- Keep a copy of your final submission. This will be helpful when you are running my tests against your final submission.
- You should begin your final submission at least 15 minutes before the due date/time to ensure you can retry if problems occur.
- Starting from two days after you receive an assignment, you are required to submit the latest version of your submission on every **even** day leading up to the due date as well as the final due date. Each version should show steady, meaningful progress. For example, if the assignment is given on the 12th, you must start your every even day submission on the 14th. If the assignment is given on the 15th, you must start your every even day submission on the 18th. Every even day does include weekends. You do not have to submit on even university holidays.
- For each assignment, you MUST turn in solutions for **all** previous assignments. For example, when you turn in Program 2, it **must** contain everything from Program 0 and Program 1. Each submission should be fully contained (i.e., the grader will not frankenstein previous and current submission together to get a working version). Code from previous assignment must be corrected; note well that code from previous assignments may be graded.
- For hard copies (only if required), you only need to print source files that are new for this current assignment. Make sure your hard copy includes all code (e.g., including code collapsed by IDE). Code should be printed landscape with two columns; format your code such that such printing does not add line breaks. Your printed code should be easily readable. Source hardcopies should be submitted in alphabetical order (case insensitive) by public class name. The package name is part of the class name (sample print ordering: pkg1.class2, pkg2.class1, pkg2.class3).

Java

- Your program must be written using appropriate facilities in the latest (as of the first day of class) release (General Availability) of Java. You may only use standard, non-preview language features.
- For JUnit testing, you must use the latest (as of the first day of class) production release of JUnit.

C/C++

- For C and C++, your program must compile with the latest (as of the first day of class) gcc/g++. You may certainly add options to change the C/C++ standard used for compilation.
- At a minimum, compile gcc/g++ with: -Wall -Wextra -Wpedantic -Wconversion
- You must test your programs with appropriate tools for memory leaks/faults, open descriptors, etc. At a minimum, you should use valgrind. Recommended options --tool=memcheck --show-reachable=yes --leak-check=full --track-fds=yes.

Specifications/Protocols

- Specifications/Protocols are written, not spoken. Make sure anything you rely on for your implementation is written in the specification or in any addendums.
- The provided specification (including any updates on Canvas) is the sole authority. Information from other sources, including the upload site, are not authoritative.
- Your protocol implementations should be robust; they should gracefully handle failures, whether accidental or malicious. Implementations must 1) assume failures are common and 2) behave defensively (i.e., input is not to be trusted). The applicable adage is that you should "be conservative in what you believe." The Heartbleed vulnerability failed in this respect because it believed that the message length in the message header was the actual message length; hackers took advantage of this misplaced trust. Implementations should also expose errors. Violations of the protocol specification should generate errors. For example, if you expect "123" and get "123ABC", it should be failure that generates an error. Note well that this may contradict a long-held Internet principle to be "conservative in what you send and liberal in what you accept." In this case, you might ignore the ABC at the end because you got the 123). For this class, we'll go with **strict protocol adherence**. Some examples of how this applies include (but are not limited to):
  - Reject messages with extraneous bytes
  - Reject messages with explicitly incorrect values. For example, if the protocol specifies that a bit must be 0, then reject if it is 1. If the protocol does not specify the semantics or values, you should not reject (i.e., just ignore). For example, if a bit is "RESERVED" but no value is specified, then you should completely ignore the value.

  Why this approach? The idea is that this may provide better security. In addition, for new protocols, a strict interpretation will help avoid implementation oddities creeping into the protocol for the sake of compatibility (e.g., We've changed the protocol to allow X because the major implementations already do X and we don't want to break things). Note that there is not general agreement on this. Some would say be "liberal in what you accept" and that such strictness 2) limits flexibility in protocol evolution and 2) in the case of inevitable protocol ambiguity may result in two complaint endpoints not being able to communicate.

