Please include comments wherever you add or modify code to explain what you’re doing. Submit your modified files on Blackboard. Links to the sample code can be found at the bottom.

1. Finish a correct implementation of the navigation system from last Wednesday’s class exercise. The sample code has been expanded a bit from the exercise so that it will compile and run using fake data (e.g. a “position” is just an int) so you can see the components interact. Enter an integer in the text box and press confirm to start. As a reminder, here is the description from the exercise:

The system consists of four components: a route calculator, a maneuver generator, an interface to a GPS unit, and a UI. The UI requests a new route by invoking a `calculateRoute` operation on the route calculator, an operation which is assumed to be computationally intensive. When finished, the route is passed to the maneuver generator via a method `setNewRoute`. The maneuver generator in turn calls `setNewRoute` in the UI so it can draw the route on the map. The GPS interface continually parses the data stream from the hardware and periodically updates the maneuver generator with the current position via `updatePosition`. The maneuver generator in turn calls `setPosition` in the UI so it can update the display; then it checks the position against the current route and generates a new turn instruction for the UI if needed (an operation which we assume is not computationally intensive), calling `announceNextTurn`.

As discussed in class the main issues to address are 1) unresponsive UI, 2) race conditions in ManeuverGen, and 3) external threads executing code in UI components. You should assume the basic design is fixed, that is, do not extensively redesign or change the relationships between components, etc., even though you may not agree with the design.

2. This is a relatively straightforward problem in using threads to parallelize a simple application. The class `Histogram` calculates letter frequencies in a character array. Use threads to execute in parallel and see how much you can speed it up. Make sure you get the same results and that there are no race conditions. (Note: if you use a thread pool from the `Executors` class, be sure you call shutdown() or else the pool threads will keep the VM alive.)

3. Finish the implementation of the `CalculatorProxy` that uses messaging. Note that the sample code depends on the ICallback and IAsyncCalculator classes from the previous examples. The basic tasks are to implement the method `public Future<Integer> increment(int n)`, and then to make sure there are no race conditions. (You can assume that only one thread will be calling the `public` methods of `CalculatorProxy`, but it has an internal thread so there is still some analysis to do.)

Code links:
http://www.cs.iastate.edu/~smkautz/cs430s13/examples/navsystem/
http://www.cs.iastate.edu/~smkautz/cs430s13/examples/histogram/Histogram.java
http://www.cs.iastate.edu/~smkautz/cs430s13/examples/messaging.zip