

Part 1

```
Embedded System Simulator
Type 'r' to reset, 'q' to quit, or any other key to provide sensor input.
Enter command: a
Enter current humidity (0-100%): 20
Enter current temperature (0-150°F): 60
=====
Current Humidity: 20%
Maximum Humidity: 20%
Minimum Humidity: 20%
Humidity Trend: N/A
Humidity Check: Low
=====
Current Temperature: 60°F
Maximum Temperature: 60°F
Minimum Temperature: 60°F
Temperature Trend: N/A
=====

Enter command: k
Enter current humidity (0-100%): 30
Enter current temperature (0-150°F): 70
=====
Current Humidity: 30%
Maximum Humidity: 30%
Minimum Humidity: 20%
Humidity Trend: Increasing
Humidity Check: OK
=====
Current Temperature: 70°F
Maximum Temperature: 70°F
Minimum Temperature: 60°F
Temperature Trend: Increasing
=====

Enter command: j
Enter current humidity (0-100%): 55
Enter current temperature (0-150°F): 65
=====
Current Humidity: 55%
Maximum Humidity: 55%
Minimum Humidity: 20%
Humidity Trend: Increasing
Humidity Check: OK
=====
Current Temperature: 65°F
Maximum Temperature: 70°F
Minimum Temperature: 60°F
Temperature Trend: Decreasing
=====

Enter command:
```

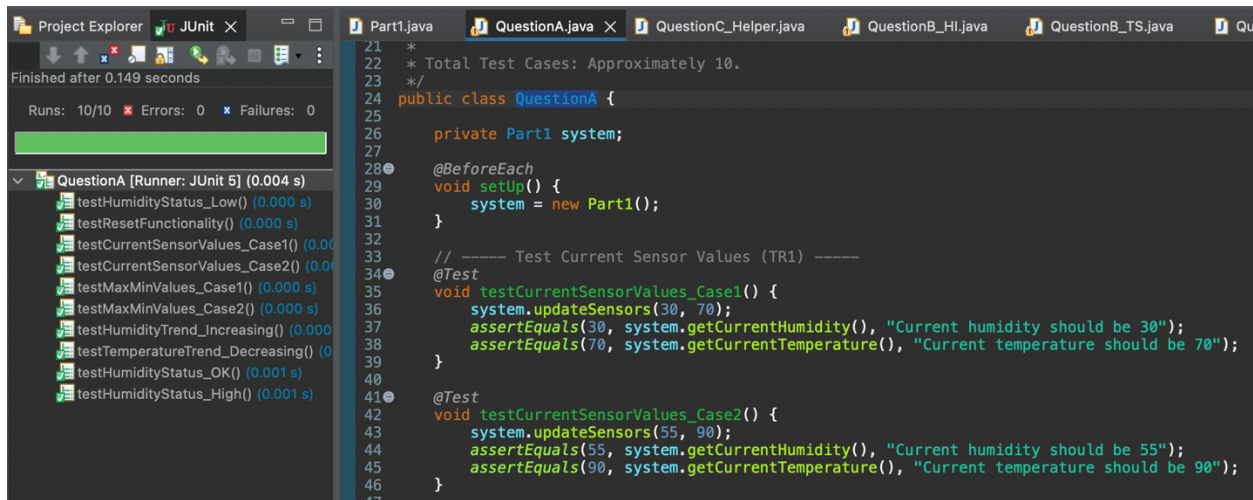
1

2

3

Part 2

Screenshots of the 9 output



The screenshot shows an IDE with two main panes. The left pane displays the Project Explorer and JUnit test results. The right pane shows the source code for QuestionA.java.

JUnit Test Results (Left Pane):

- Finished after 0.149 seconds
- Runs: 10/10, Errors: 0, Failures: 0
- QuestionA [Runner: JUnit 5] (0.004 s)
 - testHumidityStatus_Low() (0.000 s)
 - testResetFunctionality() (0.000 s)
 - testCurrentSensorValues_Case1() (0.000 s)
 - testCurrentSensorValues_Case2() (0.000 s)
 - testMaxMinValues_Case1() (0.000 s)
 - testMaxMinValues_Case2() (0.000 s)
 - testHumidityTrend_Increasing() (0.000 s)
 - testTemperatureTrend_Decreasing() (0.000 s)
 - testHumidityStatus_OK() (0.001 s)
 - testHumidityStatus_High() (0.001 s)

Source Code (Right Pane):

```
21 *
22 * Total Test Cases: Approximately 10.
23 */
24 public class QuestionA {
25
26     private Part1 system;
27
28     @BeforeEach
29     void setUp() {
30         system = new Part1();
31     }
32
33     // ----- Test Current Sensor Values (TR1) -----
34     @Test
35     void testCurrentSensorValues_Case1() {
36         system.updateSensors(30, 70);
37         assertEquals(30, system.getCurrentHumidity(), "Current humidity should be 30");
38         assertEquals(70, system.getCurrentTemperature(), "Current temperature should be 70");
39     }
40
41     @Test
42     void testCurrentSensorValues_Case2() {
43         system.updateSensors(55, 90);
44         assertEquals(55, system.getCurrentHumidity(), "Current humidity should be 55");
45         assertEquals(90, system.getCurrentTemperature(), "Current temperature should be 90");
46     }
47 }
```

Answer the following question:

- 1) What is the difference between testing the 7 inputs in a sequence and testing them individually. How are the two test cases designed? (Use narrative description, no test code needed.)

Sequence Testing (Style 1): In this approach, multiple inputs are fed consecutively into the system. The internal state (max, min, trend) is updated cumulatively. The test then verifies the final aggregated state after all 7 inputs. This method tests how the system behaves over time as new readings alter its state.

Individual Testing (Style 2): Here, each test case is isolated. The system is reset before each input, so each test verifies that the initial reading sets the current, max, and min values correctly and that the trend is “N/A” (since no previous reading exists). This approach ensures each single reading is handled correctly in isolation.

- 2) As the same person who developed, refactored, and tested the code, does your refactored code make it easier or harder to test the system, explain with examples.

The refactored code in my opinion made testing easier because it centralizes common logic into a single Question3_helper class. For example, instead of having separate code to calculate the max, min, and trend for humidity and temperature, both now use the same methods from Question3_helper. If a bug appears in trend calculation, it only needs to be fixed in one place. This reduces redundancy in both production code and tests, leading to a more efficient, maintainable and testable code.

3) Does your refactored code parametrize your tests? Explain.

The refactored code itself does not automatically parameterize tests; however, by unifying the sensor logic into `Question3_helper`, it enables the use of parameterized tests. Both humidity and temperature now share the same algorithm, allowing the same parameterized test structure to be used (in the Style 2 tests) for both sensor types without duplicating code.

4) If you received the refactored code (written by another developer) to just test it, would it be easier or harder than case iv) above?

It would be easier to test the refactored code because its modular structure isolates the core functionality. This separation allows for more focused and concise tests like we said in question 2. The common functionality is only implemented once, reducing the number of test cases needed and making debugging more straightforward. So yes, makes it easier.

5) Would you prefer to test the original code or the refactored code, if both were written by another developer?

I would prefer to test the refactored code. Its modularity and lack of duplicated logic also make it easier to understand and maintain. Any issues can be pinpointed more quickly, resulting in a more efficient testing process.