1. Calc currently implements one function: it adds two integers. Use test-driven design to add additional functionality to subtract two integers, multiply two integers, and divide two integers. First, create a failing test for one of the new functionalities, modify the class until the test passes, and perform any necessary refactoring. Repeat until all required functionality has been added to your latest version of Calc and all tests pass.
   Remember that in TDD, the tests determine the requirements. Before modifying the software, you must encode decisions such as whether the division method returns an integer or a floating point number in automated tests. Submit printouts of all tests, your final version of Calc, and a screenshot showing that all tests pass. Most importantly, include a narrative describing each TDD test created, the changes needed to make it pass, and any necessary refactoring.

   You can use any programming language to develop Calc. Please find attached Java, JS, and Python Calc implementations.

2. (Optional) You can use a continuous integration setup in GitHub. Include version control for both source code and tests and populate both with a simple example. Experiment with “breaking the build” by either introducing a fault into the source code or adding a failing test case. Restore the build.

3. (Optional) Other than automated test execution. Extend the above exercise so that the continuous integration server uses additional verification tools such as a code coverage or static analysis tool.

4. (Optional) Find a refactoring in some large, existing system. Build tests that capture the behavior relevant to that part of the system. Refactor, and then check that the tests still pass.

5. (Optional) Repair a fault in an existing system. That is, find the code that needs to change and capture the current behavior with tests. At least one of these tests must fail, thus demonstrating that you found the fault. Repair the fault and check that all of your tests now pass.