Student Name:

Unit Testing with JUnit Assignment

This assignment supports competency development and mastery. Upon completion of this assignment, students will:

- Implement multiple custom classes to match a set of behavioral and data specifications
- Incorporate a variety of data types into the classes.
- Create complete JUnit test cases to exercise the functionality of the new classes
- Create methods which follow specific application rules (business logic)

Directions:

1. Complete the steps in the assignment below.
2. From the command line at the root directory of your project execute the following command: **`git bundle create firstname-lastname-junit-date.bundle --all`** where firstname and lastname are your first and last names, and date is in the format of mmddyyyy.
3. Add the link to your GitHub PR and the bundle file created in step 2 above to your Lab 2 submission in Blackboard.

Unit Testing with JUnit Assignment:

In this assignment you will utilize the testing framework JUnit to create tests in this project.

The following section describes the behaviors and specifications for two new classes in the Java Adventure project called Lantern and TreasureChest. **You do not need to implement these classes yourself, they have already been implemented for you.** They have been created as new, separate java files, not part of any existing file. The classes have the following attributes:

1. **edu.cscc.javaadventure.Lantern**
    a. The Lantern class should represent the weight of the lantern to a precision of two decimal places. It should be able to store/represent whether the lantern is lit or unlit, as a Boolean. Lanterns should be able to be broken or unbroken, a state also represented by a Boolean. Finally, lanterns should have a string description of “A tarnished, old lantern that has seen better days.” The weight and description of the lantern should be read-only – that is to say, they can be accessed/viewed but not changed.
    b. The Lanterns should have a default constructor which initializes the weight to 1.50, ensures the lantern begins unbroken and unlit, and sets the description to the default specified above.
    c. Encapsulate the data by making the data members private whenever possible and create getters and setters for all the data members of the lantern. Provide methods to light and extinguish the lantern. These methods should set whether the lantern is lit or unlit (a Boolean variable). Provide methods to break and fix a lantern. These methods should set whether the lantern is broken or unbroken (a Boolean variable)

Assignment continues on the following page.

   d. The description of the lantern should change depending if the lantern is lit or unlit. If the lantern is lit, the description should read: “A tarnished, old lantern that has seen better days. It glows softly.” If the lantern is unlit, the description should read “A tarnished, old lantern that has seen better days. It is unlit.”

2. **edu.cscc.javaadventure.TreasureChest**
   a. TreasureChest class should represent the weight of the chest to a precision of two decimal places. It should be able to store/represent whether the chest is open or closed, as a Boolean. Chests should be able to be locked or unlocked, a state also represented by a Boolean. Finally, chests should have a string description of “A sturdy iron chest.” The weight and description of the chest should be read-only – that is to say, they can be accessed/viewed but not changed.
   b. The TreasureChest should have a default constructor which initializes the weight to 10.00, ensures the chest begins closed and locked, and sets the description to the default specified above.
   c. Encapsulate the data by making the data members private whenever possible and create getters and setters for all the data members of the chest. Provide methods to open and close the chest. Provide methods to lock and unlock a chest. Add special behavior to the setter used to open the chest. In addition to setting whether the chest is open, the setter should return a Boolean. If the chest can be opened (because it is unlocked) the setter should return true. If the chest cannot be opened (because it is locked) the setter should return false.
   d. The description of the chest should change depending if the chest is locked or unlocked. If the chest is locked, the description should read: “A sturdy iron chest. It is locked.” If the chest is unlocked, the description should read “A sturdy iron chest. It is unlocked.”
   e. The TreasureChest class should enforce the following behavior: Locking a chest that is open both closes and locks the chest. Only an unlocked chest may be opened.

Additionally, Javadoc documentation has been added to each of the classes to provide descriptions of the API (Application Programming Interface) of the two classes.

Assignment continues on the following page.

Lab Setup

1. Follow the instructions in the **Common JavaAdventure Lab Setup** document if you haven't done so already.
2. Using git, checkout the pre-existing branch from the JavaAdventure project: **`git checkout lab-2/junit-assertions-starting-point`**.
3. **Make sure you follow step 2 before following this step.** Using git, create a new branch in the JavaAdventure project: **`git branch lab-2/junit-assertions-solution`**, then **`git checkout lab-2/junit-assertions-solution`**. (Alternately you could do this in a single step with **`git checkout -b lab-2/junit-assertions-solution`**.)
4. Open the JavaAdventure project in Intellij.

Implement JUnit Tests
Your task is to create JUnit test classes, one for each class (Lantern and TreasureChest). You do not need to worry about configuring your test environment; the project is already configured for testing and is ready for you to begin working.

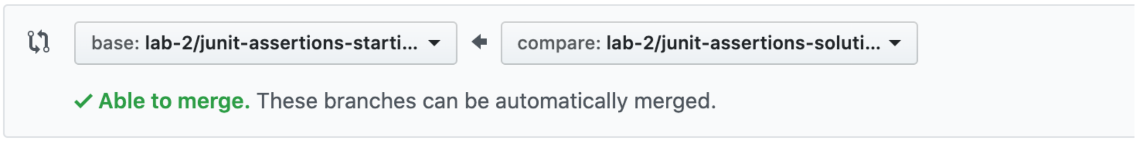
1. Right-click on the "test" folder, select New -> Java Class and create a class called **edu.cscc.javaadventure.LanternTest**.
    a. Navigate to the Lantern class and read the documentation and code to understand how the class works.
    b. Navigate back to the LanternTest class.
    c. Implement JUnit tests to test for the expected behaviors of the class listed above.
    d. Continue implementing tests until you have 100% test coverage (see the **Assignment Notes** section below).
    e. Make sure you are committing your work as you go.
2. Right-click on the "test" folder, select New -> Java Class and create a class called **edu.cscc.javaadventure.TreasureChestTest**.
    a. Navigate to the TreasureChest class and read the documentation and code to understand how the class works.
    b. Navigate back to the TreasureChestTest class.
    c. Implement JUnit tests to test for the expected behaviors of the class listed above.
    d. Continue implementing tests until you have 100% test coverage (see the **Assignment Notes** section below).
    e. Make sure you are committing your work as you go.

Assignment continues on the following page.

Pushing Your Work to GitHub

Once you are satisfied with your work do the following to submit the lab to your instructor.

1. Using git, push your branch (**lab-2/junit-assertions-solution**) to your private JavaAdventure repository.
2. Using the URL returned by the server (or from the branch on GitHub) open a Pull Request on GitHub for your branch.
    a. Fill in the PR form (title, description)
    b. Assign your instructor as a reviewer.
    c. Change the base branch to **lab-2/junit-assertions-starting-point**



base: lab-2/junit-assertions-starti... compare: lab-2/junit-assertions-soluti...

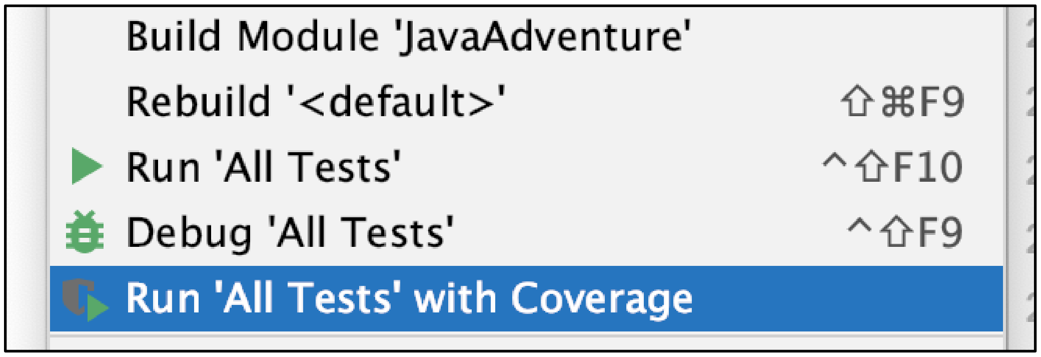
✓ Able to merge. These branches can be automatically merged.

    d. Click Create Pull Request. This will create a Pull Request.

Review ALL Assignment Notes on the following page(s).

Assignment Notes:

1. Use the “Run all with Coverage” option, by right-clicking on the “test” folder.

   This will bring up a pane with the test coverage results for the package. Double-click on the package name and it will display the test coverage for each class. As you write more tests this amount should increase. Once you have 100% test coverage for LanternTest and TreasureChestTest you will have completed the assignment.
2. Your tests should be small and focused. Typically each test should check for a single behavior. This implies that as you test for more behaviors you should increase the number of tests you have overall.
3. Commit your work frequently with git as you write tests. Committing after writing a passing test is a good practice, and it will help you to recover from changes that were incorrect more easily.