Lab 6: Conditionals, Loops, and More!

Let's get started!

Today we'd like you to:

1. **Open Eclipse**. Remember to keep all your projects in the same workspace. Think of a workspace as a shelf, and each project is a binder of stuff on the shelf. If you need some help organizing your workspace, we'd be happy to help!
2. **Create a new Java project**. Call it something that will help keep your code organized, i.e., COMP 1510 Lab 6.
3. **Complete the following tasks**. Remember you can right-click the src file in your lab 5 project to quickly create a new Java class.
4. When you have completed the exercises, **show them to your lab instructor**. Be prepared to answer some questions about your code and the choices you made.
5. **Remember that for full marks your code must be properly indented, fully commented, and free of Checkstyle complaints**. Remember to activate Checkstyle by right-clicking your project in the Package Explorer pane and selecting Checkstyle > Activate Checkstyle.

What will you DO in this lab?

In this lab, you will:

1. Evaluate boolean expressions
2. Make choices using the if and if-else statements
3. Write code that loops using the while statement

Table of Contents

1. Mathematics

The Math class is pretty handy. It has a couple of useful constants, like pi and e, and a slew of methods that perform all sorts of calculations. Let's practice implementing methods by creating our own fun version of a Math class with a little help from some unit tests:

1. Create a new class called **Mathematics** inside package ca.bcit.comp1510.lab6:
    a. Do not include a **main method** inside the class definition. Remember the main method is what gets executed first when we start our program. In this case, the Mathematics class is not a complete program.
    b. Include a **Javadoc** comment at the top of the class. The Javadoc comment should contain:
        i. The name of the class and a (very) short description
        ii. An @author tag followed by your name
        iii. An @version tag followed by the version number.
2. There is a file called **MathematicsTest.java on D2L**. Download it, and copy it to the same package. Don't edit this file.
3. When you copy the MathematicsTest.java file to your program, it will cause a compiler error. We can tell there is a compiler error because the icon for the file in the Package Explorer contains a little red square with a white X in it. If we open MathematicsTest.java, we will see **compiler complaints.**

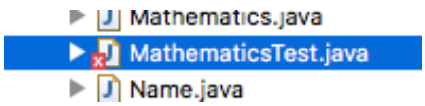

Figure 1 Uh-oh. There's a compiler error.

4. The file we copied is a **JUnit** (pronounced Jay Unit) Test Case. It uses the JUnit testing framework to test the code you will write in the Mathematics class. We have to tell Eclipse to add the JUnit framework to our project setup. It's easy.
5. Click on the first compiler error icon in the margin of the file, i.e., on line 3.
6. Choose **Fix Project Setup** (see figure below):

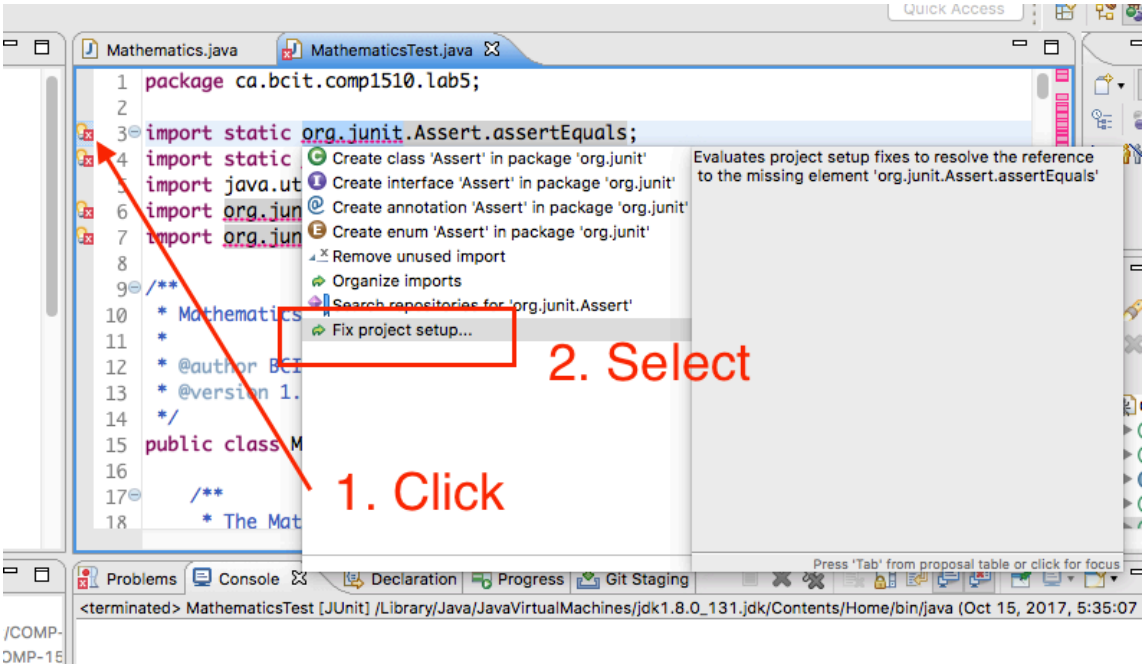

Figure 2 Click the first compiler error and choose Fix Project Setup

7. Eclipse will recommend that you **add the JUnit 4 library** to the build path. Click OK.

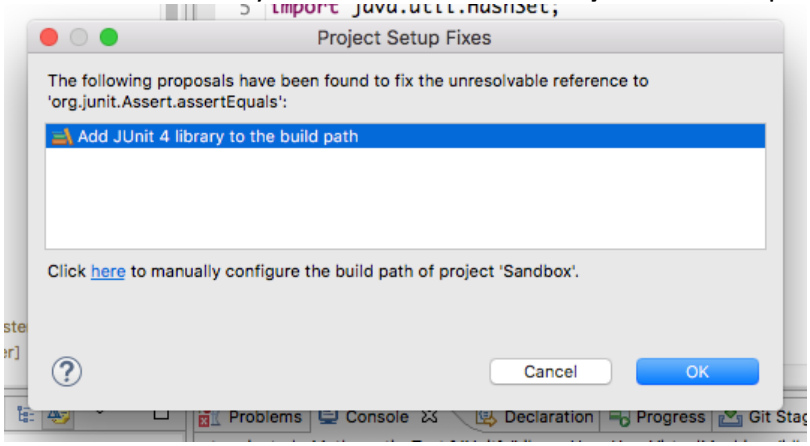

Figure 3 Add JUnit 4 library to the build path and click OK

8. Take a peek at the MathematicsTest.java file. Remember not to edit this file. It contains tests. We wrote **a collection of tests that will automatically test the methods you implement in the Mathematics class**. Before we can run the tests, we have to make sure we have created "method stubs" for all the tests in the MathematicsTest class.
9. There is a compiler error on line 38 of MathematicsTest.java. It is trying to test the value returned by the PI constant in the Mathematics class. The problem is we haven't put anything into the Mathematics class yet! Click on the error.
10. Eclipse will provide some suggestions. In this case, we want to add a constant to our empty Mathematics class. Go ahead and choose that suggestion.

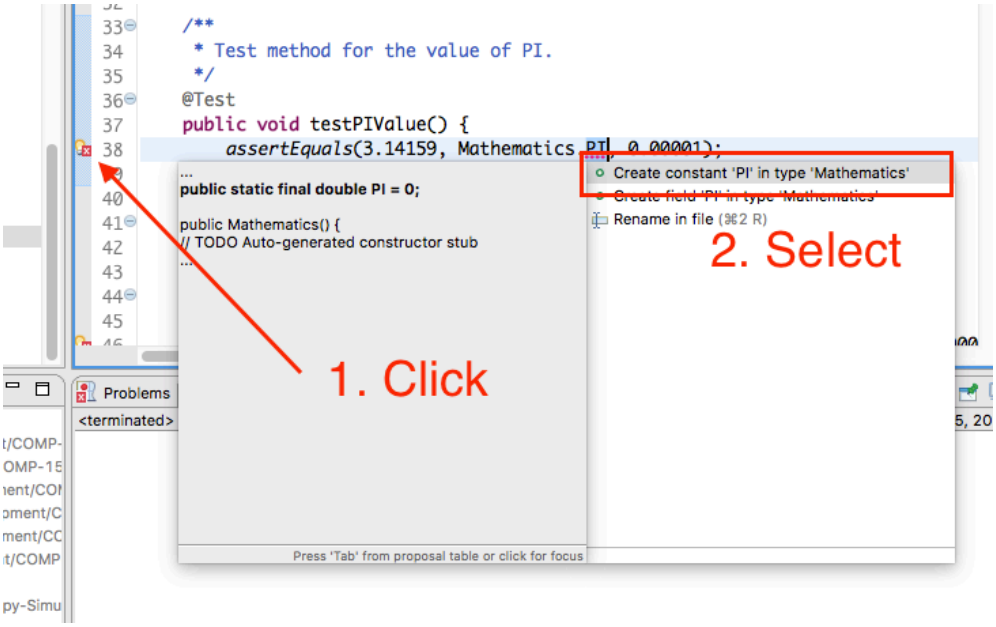

Figure 4 Click the compiler error on line 38 and select Create constant...

11. Eclipse just created a constant called PI in your Mathematics class. Don't worry about assigning the right value now.
12. There is another constant our test class wants to test on line 46. The constant is called ONE_FOOT_TO_KILOMETRE_RATIO and we need to add it to the Mathematics class. Do the same thing to add a constant in Mathematics called ONE_FOOT_TO_KILOMETRE_RATIO.
13. On line 54 of our test class, there is another compiler error. It looks like the test class is trying to test a method called getCircleArea. Click the error icon in the margin. Eclipse will suggest adding a method to the Mathematics class. Click OK.

Figure 5 Click the error on line 54 and Create a method...

14. Eclipse created an empty method in the Mathematics class. Right now, it just returns zero. You will implement it later. Save the change.
15. You will notice there is a compiler error on line 72 now. The test program is trying to test a method called getSquareArea. Click the compiler error and create a method stub in the Mathematics class again. Save the change.
16. Keep doing this until you have created a method stub in the Mathematics class and eliminated all the compiler errors. **Save all your changes**.
17. Here comes the punch line. This won't work until all the compiler errors have been resolved by making method stubs in the Mathematics class. Hold on to your hats. Run the test file. Right click on it and choose **Run As > JUnit Test**.
18. The JUnit window automatically opened and revealed the results of the JUnit tests. Each test method is run independently. Each method contains a simple test that compares the output or return value from a method with an expected value. If a method returns a wrong value, there is a X beside the test. If any tests fail, the coloured strip is red.

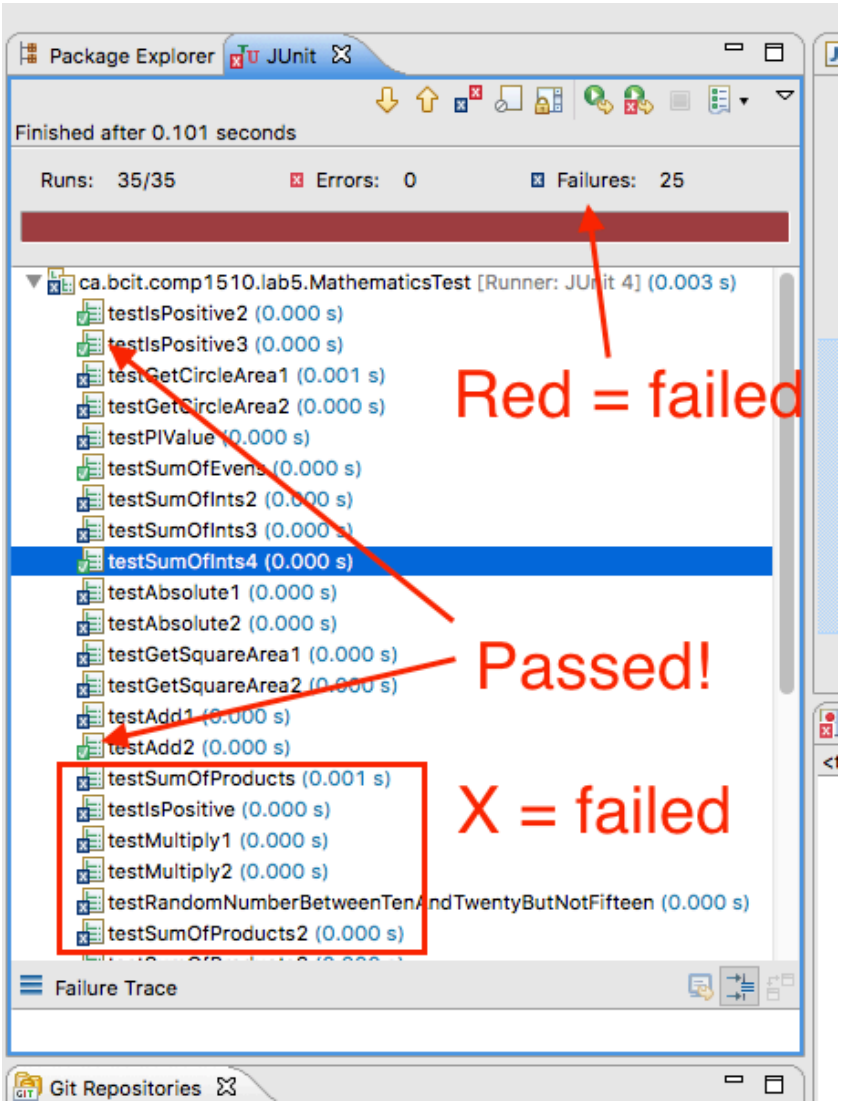

Figure 6 The results of the JUnit tests: red = FAIL!

19. Your task this lab is to implement the methods in the Mathematics class so the JUnit tests in the MathematicsTest class pass. You can click on any of the failed tests, or you can view the code in the MathematicsTest file to see how the methods are being tested.
20. When your tests all pass, the coloured strip will be green.

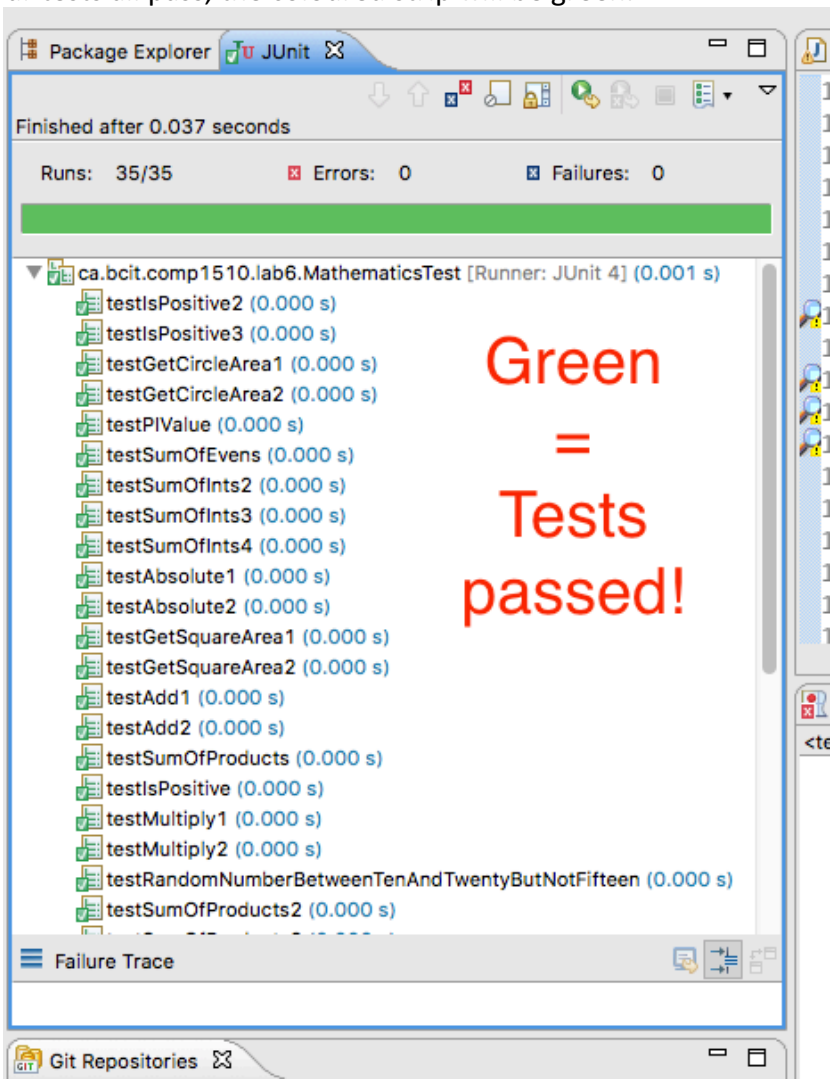

Figure 7 Green means the tests have all passed!

21. Here are some helpful Javadoc comments for you to use for each element you must complete. There are hints in the comments about how your methods should be implemented:

```
/**
 * Returns the area of the circle with the specified radius.
 *
 * @param radius
 *            of the circle.
 * @return area as a double
 */
public double getCircleArea(double radius) {

/**
```

```
     * Returns the area of the square with the specified edge length.
     *
     * @param edgeLength
     *            of the square.
     * @return area as a double
     */
    public double getSquareArea(double edgeLength) {

    /**
     * Returns the sum of the specified values.
     *
     * @param first
     *            operand
     * @param second
     *            operand
     * @return sum of the operands
     */
    public double add(double first, double second) {

    /**
     * Returns the product of the specified values.
     *
     * @param first
     *            operand
     * @param second
     *            operand
     * @return product of the operands
     */
    public double multiply(double first, double second) {

    /**
     * Returns the difference of the specified values.
     *
     * @param first
     *            operand
     * @param second
     *            operand
     * @return difference of the operands
     */
    public double subtract(double first, double second) {

    /**
     * Returns the quotient of the specified values. If the divisor is zero,
     * returns zero instead of NaN or infinity.
     *
     * @param first
     *            operand
     * @param second
     *            operand
     * @return quotient of the operands
```

```
     */
    public double divide(double first, double second) {

    /**
     * Returns the absolute value of the specified integer.
     *
     * @param number
     *            to test
     * @return absolute value of number
     */
    public int absoluteValue(int number) {

    /**
     * Converts the specified number of feet to kilometres.
     * @param feet to convert
     * @return kilometres in the specified number of feet
     */
    public double convertFeetToKilometres(double feet) {

    /**
     * Returns the sum of the positive integers between 0 and the specified
     * number inclusive. If the specified number is negative, returns zero.
     *
     * @param number
     *            upper bound
     * @return sum as an integer
     */
    public int sumOfInts(int number) {

    /**
     * Returns true if the specified value is positive, else false.
     *
     * @param number
     *            to test
     * @return true if number is positive, else false.
     */
    public boolean isPositive(int number) {

    /**
     * Returns true if the specified value is even, else false.
     *
     * @param number
     *            to test
     * @return true if number is even, else false.
     */
    public boolean isEven(int number) {

    /**
     * Returns sum of the even numbers between 0 and the specified value,
     * inclusive.
```

```
     *
     * @param number
     *            upper bound
     * @return sum of the even numbers between 0 and number
     */
    public int sumOfEvens(int number) {

    /**
     * Returns the sum of the numbers between zero and the
     * first parameter that are divisible by the second
     * number.  For example, sumOfProducts(10, 3) will return
     * 3 + 6 + 9 = 18, and sumOfProducts(10, 2) will return
     * 2 + 4 + 6 + 8 + 10 = 30 and sumOfProducts(-10, 2) will
     * return -2 + -4 + -6 + -8 + -10 = -30.
     * @param bound the upper bound
     * @param divisor the divisor
     * @return sum
     */
    public int sumOfProducts(int bound, int divisor) {

    /**
     * Returns a random number between 10 and 20 inclusive,
     * but NOT 15.
     * @return random number in range [10, 20] excluding 15.
     */
    public int getRandomNumberBetweenTenAndTwentyButNotFifteen() {
```

2. Enhancing the Name class

Now that we can use loops and conditionals we can make some changes to our Name class that will enhance its utility.

1. Copy the Name class in Eclipse from your lab 5 project to your lab 6 project.
2. **Modify the setters**. If the proposed new value (the parameter) is an empty String or if it is composed of white space only, do not assign the value to the instance variable. Do nothing (ignore the value). If the String parameter is not empty and it is not white space, assign the parameter to a local variable and format the String so that the first letter is capitalized and the rest of the letters are lower case before assigning the new String to the instance variable.
3. **Modify the constructor** too. Disallow empty Strings or Strings composed of white space. Make sure all Strings are stored in name format, i.e., first letter capitalized and the rest in lower case. If any parameter is empty or white space, assign a suitable default value to the corresponding instance variable instead. For example, if someone tries to create a Name using (“ “, “Margaret”, “”), the first name and the last name should be assigned default values like JANE and DOE.

4. Modify the **getCharacter** method so that if the user passes an integer that exceeds the length of the name, the character @ is returned instead.
5. Write a **Driver** class to prove to your lab instructor that your methods work. You can do this by creating some Names and printing their toStrings. Create correct names and names that use white space or empty Strings for their components. Mutate them, too.

3. You're done! Show your lab instructor your work.