Lab 4: Writing Classes

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**. Make it a modular project as detailed in section 1. Call it Comp1510Lab04*LastNameFirstInitial*.
3. **Complete the following tasks**. Remember you can right-click the src file in your lab 4 project to quickly create a new package or 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. Although you can cut and paste from the lab document, it will help you learn if you **type in the code by hand**. The tactile effort of typing in the code will help reinforce the Java syntax.

What will you DO in this lab?

In this lab, you will:

1. Design and implement some simple classes
2. Encapsulate your objects by using visibility modifiers and getters/setters
3. Write simple methods that accept parameters and return values
4. Implement constructors that initialize objects in logical and organized ways
5. Learn more about constructors, main methods, and final variables.
6. Write code that passes unit tests written by someone else.

Table of Contents

1. Experiment with the Integer Class

These instructions set up a modular project for the rest of the lab. You can experiment with a non-modular one if you want, get the modular one working first! Basic steps:

- In eclipse, select new Java Project
- Fill in name of project, use default directory, verify you are using Java 21 and click Next
- Check the "Create module-info.java file" box, click Finish and enter your module name for your project

Note that if you already have a project and want to make it modular, just create a module-info.java file and move libraries from the class-path to the module path. If you have a modular project and want to make it non-modular, delete the module-info.java file and move the libraries from the module-path to the class-path.

Wrapper classes are described on pages 127-129 of the text. They are Java classes that allow a value of a primitive type to be "wrapped up" into an object, which is sometimes a useful thing to do (we will see a good example of this when we study collections later in the term).

Wrapper classes often provide useful methods for manipulating the associated type. A wrapper class exists for each of the primitive types: byte, short, int, long, float, double, boolean, char. The API document for the Integer wrapper class can be found here: Integer.

1. Create a new class called **IntegerWrapper** inside package ca.bcit.comp1510.lab04:
    a. 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.
    b. Include a **main method** inside the class definition. Remember the main method is what gets executed first when we start our program. Include a Javadoc comment for the main method. The Javadoc comment should contain:
        i. A description of the method. In this case, "Drives the program." is a good idea.
        ii. An @param tag followed by args. We will learn more about parameters and arguments in a few weeks. For now, you can just write @param followed by args followed by unused.
2. Use the constants and methods in the Integer class to complete the following tasks in your class. Be sure to **clearly label your output for each task before proceeding** to the next.
3. Prompt for and read in an integer, then print the **binary, octal and hexadecimal** representations of that integer.

4. Print the **maximum and minimum** possible Java integer values. Use the constants in the Integer class that hold these values — don't type in the numbers themselves. Note that these constants are static.
5. Prompt the user to enter two integers, one per line. Use the next method of the Scanner class to read each of them in. (The next method returns a String so you need to store the values read in String variables, which may seem strange.) Now **convert the strings to ints** (use the appropriate method of the Integer class to do this), add them together, and print the sum.

2. Multisided Dice

The **Die** class in the text assumes dice have 6 sides. You may have seen dice with 4, 10, 12, 20 sides in various games. With computers we are not limited by geometric considerations and can easily create dice with any number of sides.

1. Copy the **Die** class from the lecture into package ca.bcit.comp1510.lab04. Change the name of the Die.java file and the class name to **MultiDie** (you can right-click on the Die file and use refactor>rename from the popup menu) and make whatever changes may be required for **MultiDie** to compile.
2. Modify the JavaDoc to reflect your modifications. You are an author, but the old authors are still needed unless you totally rewrite the code.
3. Remove the initialization from the **MAX** constant declaration, remove the modifier **static**, but keep it **final**. Change the name to **max**. The value will be set in a constructor.
4. Modify the **MultiDie** constructor by adding a parameter **int numSides**, and changing the body to initialize max to numSides and calling the roll method to initialize faceValue.
5. Modify the roll() method to use max instead of MAX.
6. Write a driver class **RollingMultiDice** to exercise **MultiDie**. If useful, you can start from the lecture's **RollingDice** class.
7. Answer the following questions as comments in your MultiDie class (in the class JavaDoc comments)
    1. Do you need getters and setters for **max**? Should you have them?
    2. Can you have getters and setters for **max**?
    3. Why do you think it makes sense (or not) to have max be **final**?
    4. What does **max**'s being **final** say about the abstraction of a **MultiDie** object?
    5. Is **max** an instance variable?
    6. Should you use a record for your Die? Why or why not?

3. Create a Name class

Next create a class that can be used to represent a Name.

1. Create a new class called **Name** inside package ca.bcit.comp1510.lab04 with a first name, a middle name, and a last name, all Strings (call the fields `first`, `middle`, `last`):

   a. Include a **Javadoc** comment at the top of the class. The Javadoc comment should contain:
      i. The name of the record and a (very) short description
      ii. An @author tag followed by your name
      iii. An @version tag followed by the version number.
   b. Do not include a **main method** inside the class definition. In this case, the class is not a complete program. It will simply be used to represent a Name concept.
2. The Name record should have a constructor that takes the first, middle and last names.
3. The Name record should have an **accessor** (but no **mutator)** for each instance variable. These should have names corresponding to the instance fields (`getFirst()`, etc.)
4. Create a **toString()** method which returns a String composed of the concatenation of the first, middle, and last names.
5. Write a driver class, **Names**, to test your class. Create a name from user input and print the name (using the toString() method).

4. Create a Student class that passes a JUnit test

Finally, create a class that can be used to represent a Student.

1. Create a new class called **Student** inside package ca.bcit.comp1510.lab04:
   a. 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.
   b. Do not include a **main method** inside the class definition. In this case, the class is not a complete program. It will simply be used to represent a Student concept.
2. A Student contains the following information: first name, last name, year of birth, student number, and GPA. Using sensible data types and names implied by the test code, declare private **instance variables** for each of these.
3. A Student has a public constructor which accepts one parameter for each of the instance variables. The body of the constructor should assign each parameter to its respective instance variable (you can wait to create this constructor and methods until the next steps after using the test file to help).
4. Student has an accessor and a mutator for each instance variable.
5. Student has a toString() method which returns a String composed of the concatenation of the information in the Student.
6. There is a file called **StudentTest.java** on D2L. Download it and copy it to the same package in eclipse (you can use the OS copy and eclipse paste to achieve this). Don't edit this file.
7. When you copy the StudentTest.java file to your program, it will cause compiler errors. We can tell there are compiler errors because the icon for the file in the Package Explorer contains a little red square with a white X in it. If we open StudentTest.java, we will see **compiler complaints.**

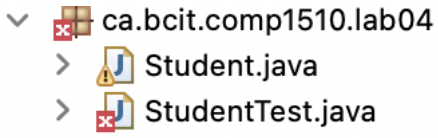

Figure 1 Uh-oh. There's a compiler error.

8. 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 Student class. We must tell Eclipse to add the JUnit framework to our project setup. It's easy.
9. Open the StudentTest file in Eclipse and **Click** on the first compiler error icon in the margin of the file, i.e., on line 3.
10. Choose **Fix Project Setup** (see figure below):

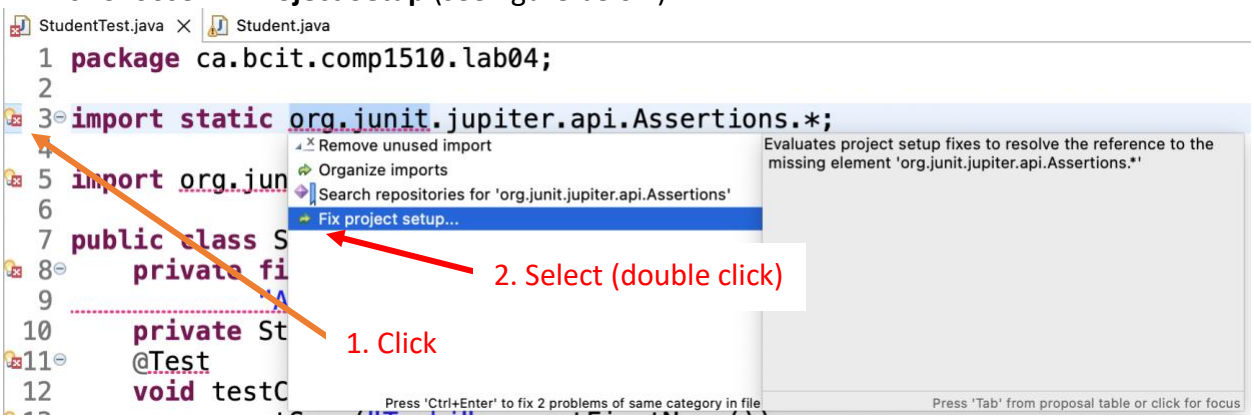

Figure 2 Click the first compiler error and choose Fix Project Setup

11. Eclipse will recommend that you **add the JUnit 5 library** to the build path. Click OK.

12. Look at the StudentTest.java file. Remember *not* to edit this file. It contains (simple) tests. We wrote a collection of tests that will automatically test the methods you are to implement in the Student class. Before we can run the tests, we must make sure we have created "method stubs" for all the tests in the StudentTest class.
13. On line 8 of our test class, there is a compiler error. It looks like the test class is trying to use a constructor. Click the error icon in the margin. Eclipse will suggest creating a constructor in the Student class. Click OK.

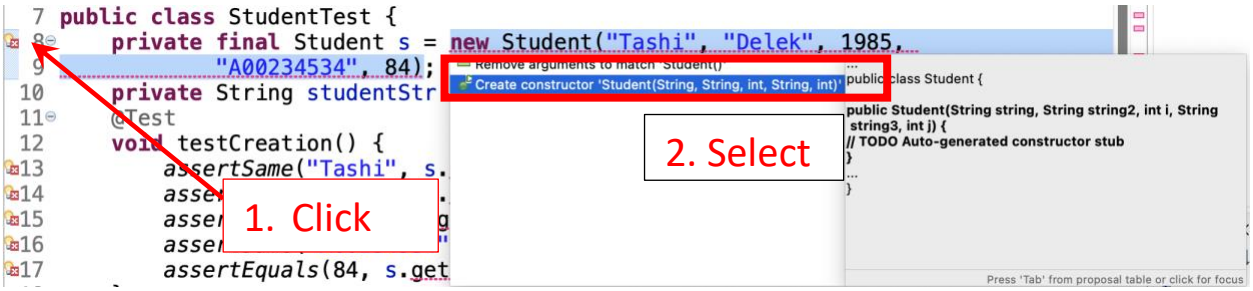

Figure 3 Click the error on line 8 and Create a method...

14. Eclipse created an empty constructor in the Student class. Right now, it does nothing. You will finish it later (including adjusting parameter names). Save the change.
15. You will notice there is a compiler error on line 13 now. The test program is trying to test a method called getFirstName. Click the compiler error and create a method stub in

the Student class again (you can also hover over the error). Save the change (you will modify the return type to String later).

16. Keep doing this until you have created a method stub in the Student class and eliminated all the compiler errors. **Save all your changes**.
17. Here comes the punch line. After all compiler errors have been resolved by making method stubs in the Student class, run the test file. Right click on StudentTest 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.

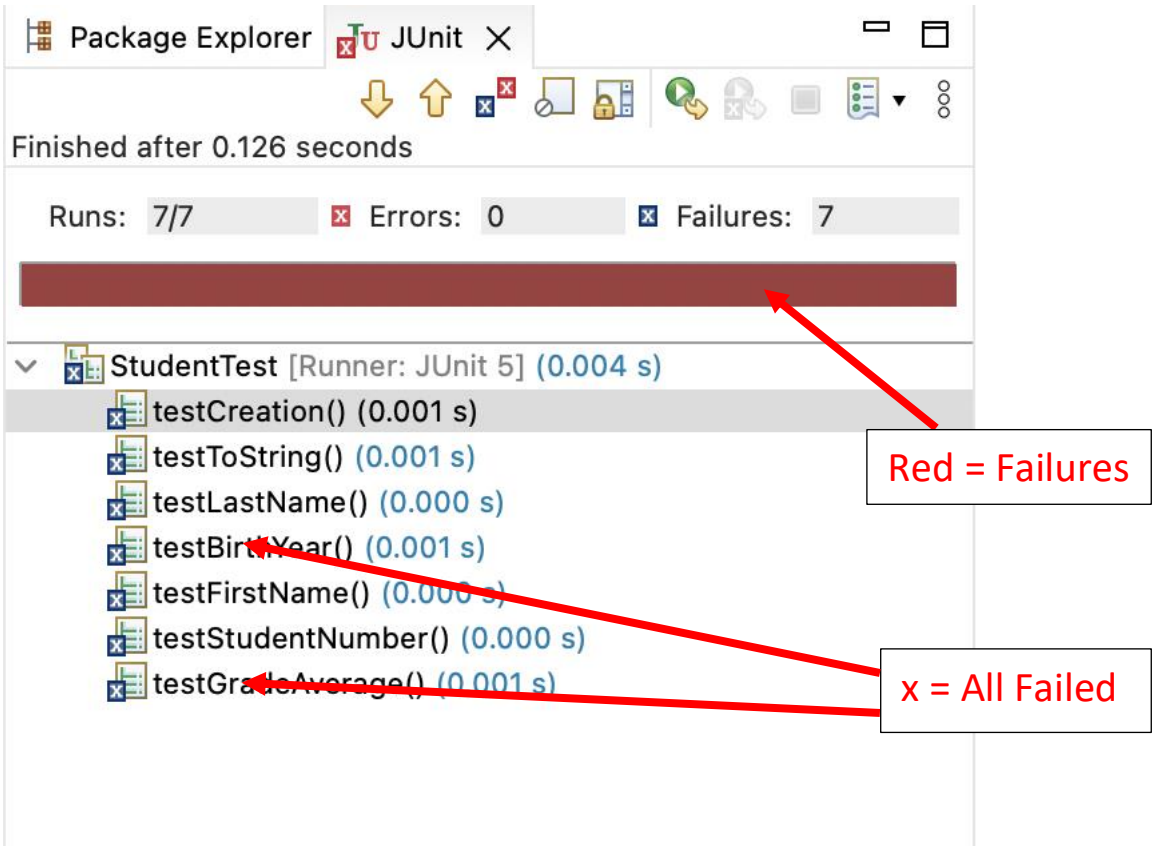

Figure 4 The results of the JUnit tests: red = FAIL!

19. Your task this lab is to implement the methods in the Student class so the JUnit tests in the StudentTest class pass. You can click on any of the failed tests, or you can view the code in the StudentTest file to see how the methods are being tested.

20. When your tests all pass, the coloured strip will be green:

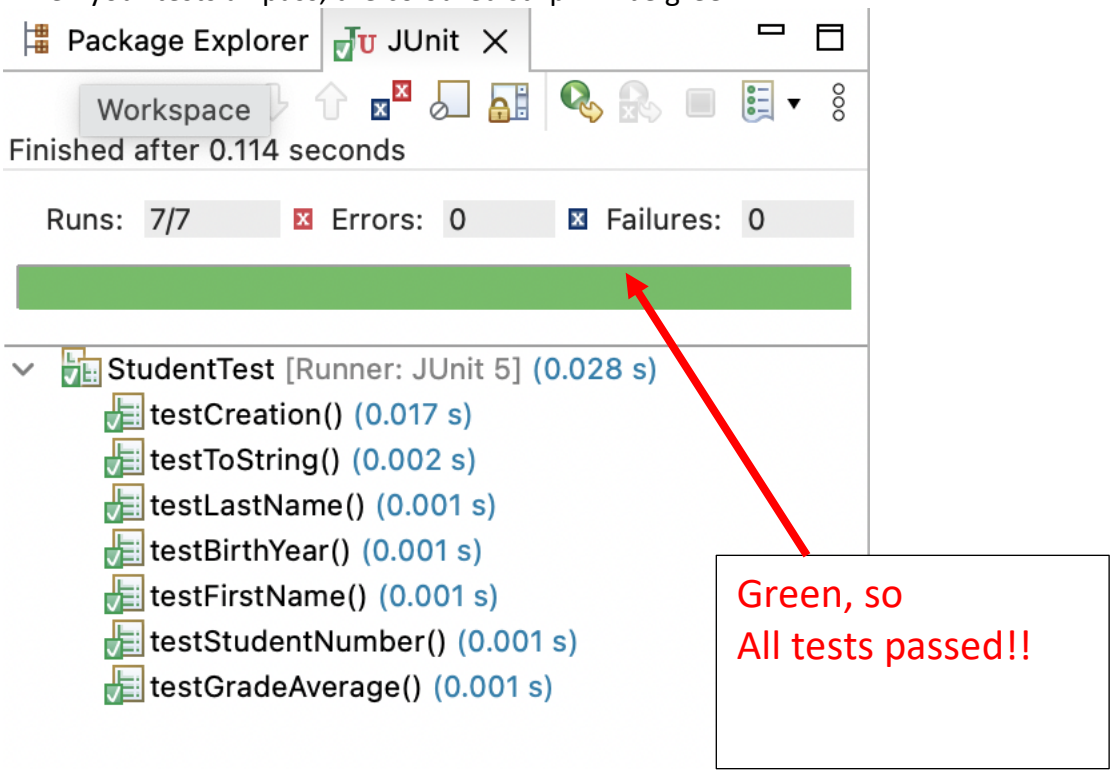

Figure 5 Green means the tests have all passed!

21. We did not need to write a test driver class, as there is a unit test class.

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

If your instructor wants you to submit your work in the learning hub, export it into a Zip file in the following manner:

7. Right click the project in the Package Explorer window and select export…
8. In the export window that opens, under the General Folder, select Archive File and click Next
9. In the next window, your project should be selected. If not click it.
10. Click *Browse* after the "to archive file" box, enter in the name of a zip file (the same as your project name above with a zip extension, such as Comp1510Lab04BloggsF.zip if your name is Fred Bloggs) and select a folder to save it. Save should take you back to the archive file wizard with the full path to the save file filled in. Then click Finish to actually save it.
11. Submit the resulting export file as the instructor tells you.