CS6312
Lab 03

Automated Testing using JUnit

Objectives:

- Implement unit tests using the JUnit testing framework
- Write JUnit tests to analyze expected results to confirm correct functionality

Overview:

This week we'll focus all of our attention on unit testing with JUnit. Therefore we won't develop a working executable, but instead will incrementally develop and test a model class – `BikeLock`.

The `BikeLock` class will be used to model a 'standard', free-hanging, 4-digit bicycle lock like the one shown here. The idea behind the `BikeLock` is that there are 4 dials, each with the numbers 0 through 9. In order to open the lock, the user must turns the dials in such a way as to make the correct numbers show at the correct locations. Once the numbers are aligned correctly, the lock can be pulled and will open.

For this exercise, you'll develop our methods one at a time, writing a series of JUnit tests after each method to confirm correct functionality. We'll get you started on the first few, but then leave the rest of the testing to you. Be sure that you are applying appropriate style guidelines to your code and committing to your repository as you go. We've included a few reminders about committing in this document, but remember any time you have included new functionality you should be committing (and writing a descriptive commit message to explain what's new).

What to do:

1. Start a new Eclipse Java project with the name `Lab03`

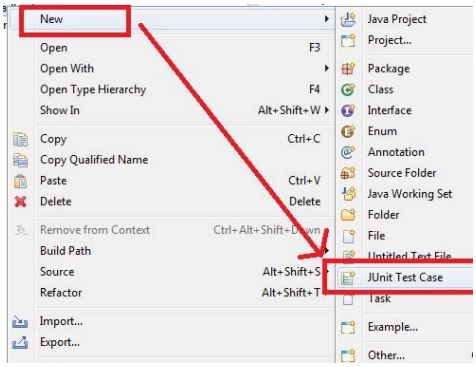
2. Create a package named `edu.westga.cs6312.locks.model`

3. Add a class named `Dial` that contains

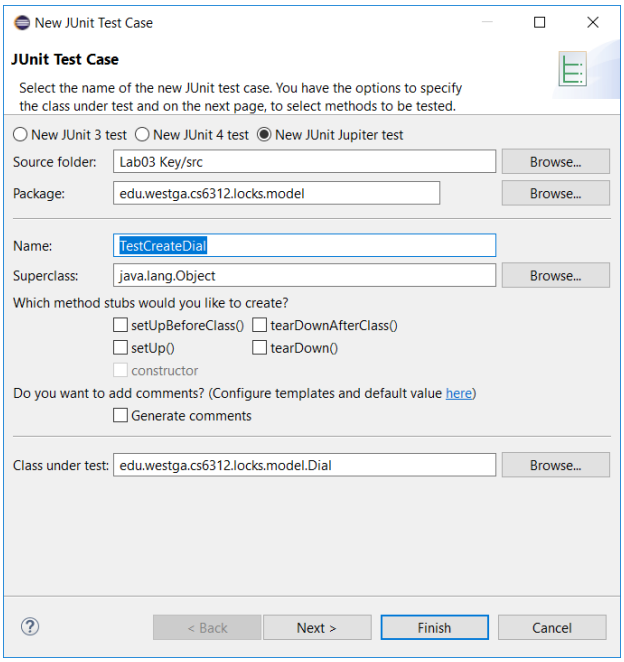
    a. Instance variables for:
        i. Current value
    b. A static final constant, `MAX_VALUE`, that is set to the `Dial`'s highest value (9)
    c. Methods:
        i. A 0-parameter constructor that sets the current value to 0
        ii. A method named `getValue` that will return the current value of this `Dial`

4. We have some code written now, but how do we know it works? Of course the easy answer is, "Well, it's pretty basic code, I can just look at it and know it's correct." Fair enough. But (1) this won't always be the case and (2) we're learning about JUnit this week; therefore we're going to go ahead and write some code to test this. In previous exercises, you've written code that would either print to the screen some output confirming that it works, or you might have even written a small, throw-away program to test it. But can you imagine how bulky / awkward this would be if you had a substantial application with lots of moving parts? The automated testing will help us with that. As with most things, there are many ways to do this stuff. Feel free to use whatever method you like, but please know that this way is confirmed to work.

   a. Go to the Package Explorer window and right-click on the Dial class (because this is the class we're going to write the tests for) and choose **New > JUnit Test Case**



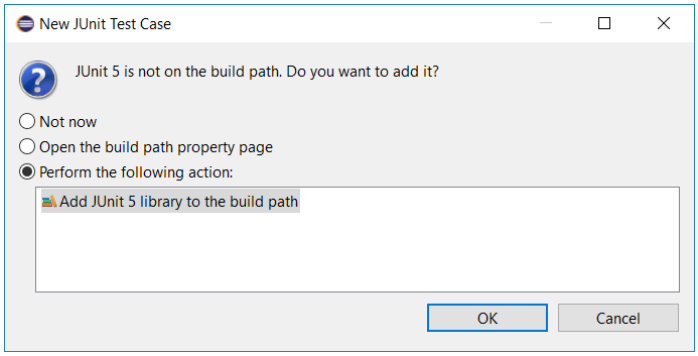
   b. In the New JUnit Test Case dialog, be sure:



   i. **New JUnit Jupiter test** is selected at the top
   ii. Modify the name (this will be the name of the new JUnit test class) to be `TestCreateDial`. Note that in general, we will always try to keep our classes organized by using the naming scheme
   Test <description of what's being tested)
   iii. The Class under test is `edu.westga.cs6312.locks.model.Dial` (if it isn't listed, you can always type in the class name, `Dial`, and click the **Browse** button to locate the class to be tested)
   iv. Click **Next**

c. The next dialog will give you an opportunity to select which method in the class you want tested. Know that you aren't required to do this step, but if you do, Eclipse will generate some code for you. If you're interested, click the box next to `getValue` to select it. Click the **Finish** button.

Note: Because this is the first JUnit test added to our project, you will be presented with a dialog asking you if you want JUnit 5 to be added to the build path. Be sure that it is being added (otherwise you can't use JUnit!) and click **OK**

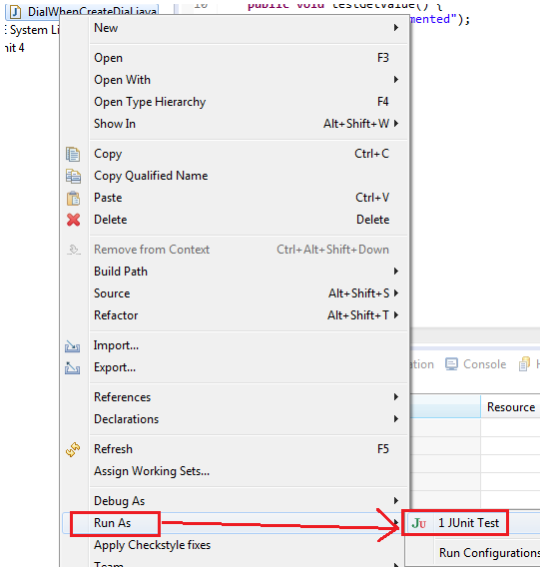


Eclipse will now generate a new Java file named `TestCreateDial`, along with some code.

d. Unfortunately Eclipse is going to cram this new test class into the same package as our Dial class, which we don't want. Instead, let's include better organization of our code by creating a new package named `edu.westga.cs6312.locks.testing.dial`

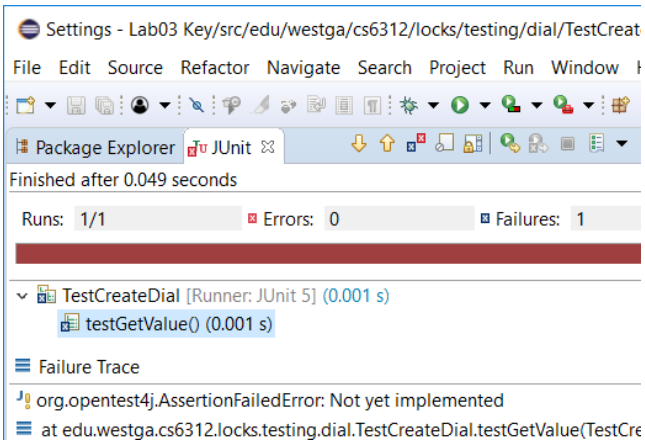
e. Once you've created this new package, go to the Package Explorer window and drag the new test class over and drop it in the testing package.

5. While the test stubs (a method with a minimal amount of code inside) are nice, you'll see that initially they fail. Go ahead and see what happens when a test fails by running this class. Go to the Package Explorer, right-click on the test class name, and choose **Run as > JUnit test**



Doing this will add a new tab next to the Package Explorer named JUnit. The test results show there.

As you can see, the test ran, but failed. Notice that the bottom area of this tab is named Failure Trace and provides details about what went wrong.

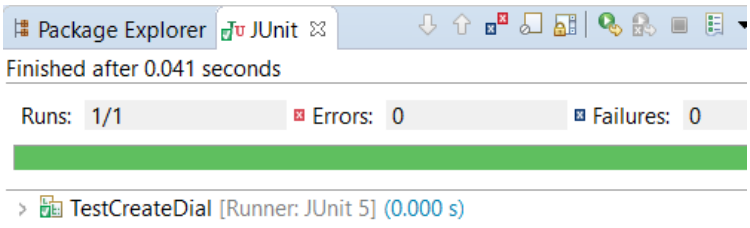


6. The next step is to add code to our test class to make the test pass. The nice thing about the JUnit package is that they provide a class named `Assert`, which has method to be used to check to see if two values match. There are many methods for us to use, but for this course we'll ask that you only use the `assertEquals` method. The idea with `assertEquals` is that we will first list the value we are expecting, and then list a call to the method we are testing. For this test, we're checking to make sure that the `getValue` method is returning the correct initial value. Therefore we'll need to:

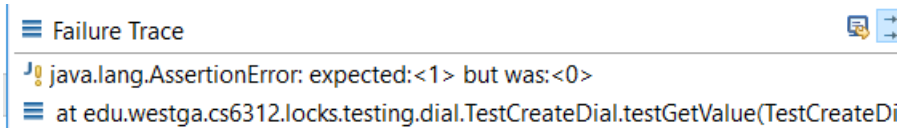
    a. Create an object of the `Dial` class
    b. Use that object to call `getValue` and store the result into a local variable
    c. Include an `assertEquals` statement to confirm that the two values are equal

```
/**
 * Test to see if the 0-parameter constructor creates
 *  a Dial with current value 0
 */
@Test
public void testGetValue() {
    Dial newDial = new Dial();
    int value = newDial.getValue();
    assertEquals(0, value);
}
```

7. After typing the code for this test, go ahead and re-run the test to confirm that the results are correct.



8. Please note that the order of values in the `assertEquals` statement is important. That is, we want to list the expected value first. And the reason why is because if your test should fail, then the message inside the Failure Trace section of Eclipse will help you. To see this, go ahead and change the expected value from 0 to 1 and re-run the test. You'll find that the Failure Trace section looks something like:



That is, this section is telling us that our method failed and the reason why was because the test was *expecting* to see 1, but the value that it actually received from calling the method was 0. But this only works if you actually set up the `assertEquals` statement properly. Again, we're just making sure that the expected value is listed first to help make our lives easier. Please do things properly.

9. Once you're done playing with the Failure Trace area from the previous step, go ahead and change things back so that your test passes. Now that you have some functionality in your Dial class *and* you've tested it to confirm that it works, now is an excellent time to make a commit to your repository

10. OK, now I'll admit at this point, the testing looks pretty silly. We wrote extra code to confirm something obvious worked. Let's try to go to the next step where we're actually manipulating the object. Go back to the `Dial` class and add a method named `increment` that will not accept or return any values. It will be used to increment the `Dial`'s current value. Remember that if the initial value is at the maximum (9) and this method is called, the new value will be 0.

11. Once you method is written, go ahead and create a new test class `TestIncrementValue`. Inside this class you'll want to create a test method `testIncrementNewDialShouldGiveValue1` (see how descriptive this is – please do this, because it will make troubleshooting later so much easier):

```
/**
 * Test to confirm incrementing a new Dial will produce
 *  a current value of 1
 */
@Test
public void testIncrementNewDialShouldGiveValue1() {
    Dial testDial = new Dial();
    testDial.increment();
    int value = testDial.getValue();
    assertEquals(1, value);
}
```

Now some of you may look at this and think, "Hmmm… but what if something strange happened and that `testDial` object didn't actually start at 0? I mean, shouldn't we assert that it was 0 originally?" This is a great point – but we don't really need to assert this first because we have a test (from before) that tests this very thing. Sure, it's in a different test class, but we've already tested it. This is also a good place for me to note that we want to avoid placing more than one `assertEquals` statement in any test method (because if the test fails, which assert failed?)

Save and run your test and be sure it passes before moving on.

12. OK, so now I would ask you, "Does this mean that your increment method is working exactly correct?" If you stop and really consider the code inside of the method, you'll see that there is a tricky spot. And actually this is a good time to talk about the number of tests to be written. There's no hard-and-fast set number of tests to be written, but instead you'll always want to think about the code and at least consider (a) the boundaries and (b) the 'sunny-day' areas. In our example here, the boundaries will be the 'edges' of the Dial's numbers – the minimum (0) and maximum (9). A 'sunny-day' value would basically be anything in between.

    So now it looks like we've already tested one boundary condition – when the Dial is at 0. Let's turn out attention to a 'sunny-day' case. You could really do any of the values here between 0 and 9, but myself, I prefer to take something as close to the middle as possible, so let's use 5. You'll need to increment the Dial to get it up to 5, then call increment 1 more time and assert:

```
/**
 * Test to confirm incrementing a Dial from 5 will produce
 *  a current value of 6
 */
@Test
public void testIncrementDialAt5ShouldGiveValue6() {
    Dial testDial = new Dial();
    for (int count = 0; count < 5; count++) {
        testDial.increment();
    }

    testDial.increment();
    int value = testDial.getValue();
    assertEquals(6, value);
}
```

    Save and run your test and be sure it passes before moving on. Notice that in order to run this test, you'll actually run both tests in this class. That's a good thing because it's really easy to run these tests. It's also the reason why we beg you to name your test methods appropriately. Can you imagine if you had a test class with hundreds of test methods and one fails? If that test method was named `test59`, you'd have no idea what broke, but if you gave it an appropriate name, it would easily help give you a clue as to what happened.

13. Now it's time to test the other boundary. Write a new test method to confirm that incrementing a `Dial` at 9 will result in the value 0. Save and run your three tests (the two boundaries and the sunny-day test).

14. OK, so it seems that things are great, right? But think about a real lock like this. What if someone tries to increment more than 10? What happens in real life? Does your code function this way? Go ahead and write a test to confirm this is true. Once they all pass successfully, go ahead and save your work and commit to the repository.

15. Now that you have a `Dial` class with a constructor, `getValue`, and `increment` methods that you know work (because you tested them completely), there's only one thing left to do: Write a method `decrement` that will not accept or return any values. It will be used to decrement the `Dial`'s current value. Note that if the initial value is at 0 and this method is called, the new value will be the maximum (9).

    Once you've completed this method in the `Dial` class, go ahead and make a new test class with four test methods inside (two boundaries, one sunny-day, and one 'wrap around' test). When you are done with this, you're finished with the `Dial` class. Go ahead and commit to your repository.

16. So now it's time to turn our attention to the `BikeLock` class. Armed with our `Dial` class, this one shouldn't be too tough. But again, our goal in this exercise is to learn about and practice JUnit testing. So we'll continue to model the process of iterative development and do things one step at a time.

    Start by adding this new class to the `model` package. It should define two `ArrayList` instance variables: one that will store four `Dial` objects (the lock itself) and a second that will store four integer values (the lock's combination). Go ahead and write a 0-parameter constructor that will store objects into each of these `ArrayLists`. Don't forget that instantiating the `ArrayList` just means that this object is *capable* of holding a series of objects. You'll need to instantiate each object inside individually as well.

    Now, because this is a 0-parameter constructor, go ahead and set the combination to 1234. Once you have completed this, it's time to test our code. OK, but we don't have any accessor – no way to actually pull the data out of the `BikeLock` object. To help with this, let's include a `toString` method inside our `BikeLock` class. This method is going to return a `String` in the format:

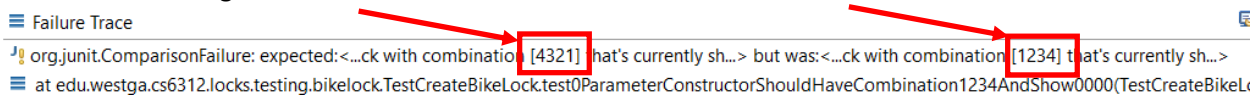
    ```
    BikeLock with combination XXXX that's currently showing YYYY
    ```

    (where XXXX is the four digit lock combination and YYYY is the four digits currently showing on the lock).

17. Now it's time to make our test class. We'll follow the convention and give this the name `TestCreateBikeLock`. Be sure to create a new package for this test class (`edu.westga.cs6312.locks.testing.bikelock`) Since we only have a 0-parameter constructor and no other mutator methods, we really only have 1 test to write. That is, we'll want to assert that the lock's combination is 1234 and that it is currently showing 0000.

```
/**
 * Test to be sure that the BikeLock's 0-parameter constructor will
 *  correctly create a lock with combination 1234 and have four dials
 *  each showing the value 0 (for a current display of 0000)
 */
@Test
public void test0ParameterConstructorShouldHaveCombination1234AndShow0000() {
    BikeLock theLock = new BikeLock();
    assertEquals("BikeLock with combination 1234 that's currently showing 0000",
            theLock.toString());
}
```

Recall that the Failure Trace area is there to help you out when your tests don't pass. With String values, the message will use [ ]'s around the characters that are different. For example, I modified my test to show a combination of 4321 as the expected value. When I run that test, the message shows as:



Once you have the constructor / `toString` combination passing this test, go ahead and save your work and commit the changes to the repository.

18. I'm sure you realize that having a combination like 1234 is pretty lame ("That's the kind of thing an idiot would have on his luggage!"), so let's go ahead and write another constructor that will accept four (4) integer values to be used at the combination. Once you have the code written, go ahead and add three new test methods (two boundaries and a 'sunny day') to your test class.

19. Hopefully when you wrote this new 4-parameter constructor you saw the code duplication inside the 0-parameter constructor. No problem, just go back and modify your 0-parameter constructor so that it is calling on this new 4-parameter constructor (this is called *constructor chaining* keep this in mind, we'll be using this term later in the semester). Because we modified our existing code, that used to pass testing, it's worth retesting that code again. This is known as *regression testing* and it's something that we'll always be doing as we go along – especially before you submit your work for grading. Remember, any test that fails means that there's something unexpected in the code and needs to be fixed. Of course the beauty of automated testing is that really all we'll ever need to do is to click a button in Eclipse and re-run the tests. The great thing is that if you select the package name (in the Package Explorer window) and then run the tests, Eclipse will run *all* of the tests in that package. How cool is that? Now it's super easy to run all of the tests and check to see that they all pass.

20. Complete this exercise by including (and testing!) `BikeLock` methods:

    - `void incrementDial(int, int)` – this method will increment the `Dial` at the position specified by the first `int` value passed in (this should be 0-based, so the first `Dial` is at position 0) the number of turns specified by the second `int` value. For example, if the first `Dial` was currently at position 4 and the call was made `incrementDial(0, 3)`, then this first `Dial` would end up at position 7

    - `void decrementDial(int, int)` – this method will decrement the `Dial` at the position specified by the first `int` value passed in, the number of turns specified by the second `int` value

    - `boolean isOpen()` – this method will return true if the `Dial`s are currently showing the same numbers, in the same order, as the combination

    Remember that each `BikeLock` method should have its own appropriately named test class, with enough methods to be sure that the code is functioning correctly.

    Please don't forget to practice incremental development by writing a little bit of code, saving, completely testing the code, and committing to your repository before you move forward to the next little bit of code.

Submitting:

When you are finished, make one final commit to your repository, be sure that all of the class files are saved, close Eclipse, and Zip the folder holding your Eclipse project using 7-Zip. Give your file the name 6312*YourLastName*Lab03.zip. Upload the Zip file to the appropriate link in Moodle.

It's probably worth taking a moment to download the file you uploaded to Moodle and double-checking to be sure that it contains everything you are required to have (including the repository).

As a final thought, it's probably worth explicitly noting that when you finish this exercise you will *not* have a working application. Instead, you'll just have two classes that have been completely tested. And while I doubt any end users will ever pay you for such a thing, we have laid the groundwork for other applications. That is, now we have a couple classes that we are assured work properly (because we tested them) and those classes can be placed into any application without fear of then having bugs.

Also know that this is how we will be developing our applications going forward. That is, when you sit down to write an application, you'll always start by writing out the model classes (one method at a time) and testing them (with multiple tests on each method, as they are completed). And then only turn to the user interface once the model classes are completed. This is sure to make your development life far less frustrating – at least that's why programmers do things this way.