JUnit Tests

How did you test your code in CSCI 150 (or whatever was your last programming class)?

A. I typed in the test cases I was given. If my program ran correctly with those it must have been okay.
B. I also tested extreme cases -- what happens if you get unexpected input.
C. I added print statements to my code to make sure that I got to certain parts of the code.
D. I was supposed to test my code?

Many software engineers divide software testing into 4 levels:

Unit testing tests whether the individual functions and methods of a program work correctly.

Integration testing tests whether a group of methods work together correctly. For example, if you are building a game integration testing will test that different characters interact correctly.

System testing tests whether an entire program works.

Acceptance testing tests whether a program meets its specifications.

Commercial software – something like Microsoft Word or an aircraft guidance system or the Banner system that holds all of the data at Oberlin College – is often hundreds of thousands or even millions of lines of code. For comparison, a full 32-volume set of the *Encyclopedia Britanica* has about 1.6 million lines of text.

Testing commercial software is a major task. Compared to that, the small programs we will write in this class are relatively easy to test. We will focus on unit and integration testing – do our individual functions work correctly by themselves and do they work together correctly?

JUnit is a library that helps with unit testing in Java. In practice it is the most commonly used external Java library.   The old way to test was to insert a bunch of print statements into your code to check the values of variables, then to run your program with specific inputs to see if it printed what you expected. This suffers from several problems:

- It is slow -- you have to insert the print statements then remove them when you are done.
- You have to re-enter the input each time you run a test.
- It is sometimes not clear if your program passed or failed a test.
- You have to have a running program before you can start testing.

JUnit is designed to address these problems. Rather than modifying your code it creates a new class for testing. You enter your test cases once and run the test suite until your code passes all of the tests. The JUnit tests include *assertions* about the state of your program -- whether two things are equal, whether some condition is true, and so forth. If any of these assertions is incorrect when your program runs, it fails the test.

Perhaps most importantly, you can do a Junit test on a single procedure; you don’t need a working program

For example, to test your MyArrayList class you might create two variables

```
MyArrayList<Integer> test = new MyArrayList<Integer>();
ArrayList<Integer> real = new ArrayList<Integer>();
```

The *test* variable and the *real* variable should behave identically so you can go through a series of steps and assert that their behavior is the same:

```
test.add(5);
real.add(5);
assertEquals("Size after addition", test.size(), real.size());
assertEquals("First addition", test.get(0), real.get(0));
```

and so forth.

You should use Junit for Lab 2. If you find that you don’t like it, here is an alternative for future labs: You will spend most of your time this semester building classes. Try to keep your classes in running order – constructors working, no undeclared variables, and so forth. Also keep a Test class, which only has a main method. Every time you add a method to your code write a method for your Test class that makes one or more objects from the class and tests whether the new method is working properly. The body of your main method just calls these Test methods. This is not shorter than using   Junit and it is probably not as reliable, but it is better than not testing at all.

The bottom line is this: you must test. It is faster to test each method just after you write it than it is to write a whole suite of methods or a whole class and then test. The smaller the chunk of code you are testing, the easier it will be to correct. Testing doesn’t take much time; finding the real problem once you know something is going wrong is what takes time. Unit testing will help you to not have to spend your time looking for bugs.

Some of our alums who are working software engineers (e.g. Ted Warner at Google) use *test-driven development* – when they need to write a chunk of code they write all of the tests for it before they write any code. This has two advantages:

- It reduces *Confirmation Bias* – you know that the code you just wrote is correct so you write tests to confirm that it is right rather than tests to find things that are wrong.
- To write tests you have to think about what could go wrong. If you design the tests first and then write the code thinking about what could go wrong, you are much more likely to write the code correctly.

The people who use it say that test-driven development speeds up the whole development cycle. Think about that.

Every bit of code that you write for every CS class you take will be graded according to whether it runs correctly. Whether you write your tests before or after you write your code, you need to test your code thoroughly before you hand it in. Find a way that works for you to incorporate testing in your coding habits.