JUnit Testing

Warm up

1. Determine the equation of the linear function from the given points.
2. How do you know your equation from #1 is correct? What are some ways you could check your answer?

x	f(x)
0	3
2	-1
3	-3
-2	7
-4	11

Checking for Correctness

f(x) = -2x + 3

f(0) = -2(0) + 3 = 3
f(2) = -2(2) + 3 = -1
f(3) = -2(3) + 3 = -3
f(-2) = -2(-2) + 3 = 7
f(-4) = -2(-4) + 3 = 11

x	f(x)
0	3
2	-1
3	-3
-2	7
-4	11

Similar thing in programming?

f(x) = -2x + 3

f(0) = -2(0) + 3 = 3
f(2) = -2(2) + 3 = -1
f(3) = -2(3) + 3 = -3
f(-2) = -2(-2) + 3 = 7
f(-4) = -2(-4) + 3 = 11

```
int f(int x) {
   return -2*x+3;
}
assertEquals(3, f(0));
assertEquals(-1, f(2));
assertEquals(-3, f(3));
assertEquals(7, f(-2));
assertEquals(11, f(-4));
```

What is unit testing?

- **Unit tests**: software tests of an individual software unit, such as an individual class or individual method
- Normally practiced by developers who write tests of individual program units to check their behavior before releasing them for inclusion in a larger project

What is the purpose of testing?

- Running software tests can *never prove that software is bug-free*.
- Can only prove *a bug exists when a test fails*.
- Purpose of testing is to demonstrate that there are bugs so that they can be fixed.
- Passing multiple tests does not guarantee that your code works correctly in all situations but gives you some confidence that it *works in common cases*.

What is JUnit?

- **JUnit** is a class library for writing unit tests for Java software.
- JUnit allows us to write software tests in the form of code (i.e. writing code to test code)
- Why write code to test code?
  - Easier and faster than testing by hand
  - Can automate testing, making it even easier/faster
  - Eliminates error in human testing

What's in a test?

- **Assertion**: An expression of an expected outcome.
- Each **test method** uses one or more *assertions* to *express* the outcome.
- Assertions must *check something* and confirm the desired effect has been achieved.
- Usually, that means a test will manipulate one or more objects, and then *confirm* that these objects ended up in the desired state.

Writing an Assertion

- assertEquals(*expected, actual*);
- assertTrue(*expression*);
- assertFalse(*expression*);
- assertEquals(*d1, d2, tolerance*);

Each "test" is a method

- **Test method**: each "test" is phrased as a method within the test class; serves as a "check" for a particular feature or behavior.
- Most think of test methods as *tests* or *test cases*
- Naming convention: Each test method name should start with "**test**" and must be preceded by the **@Test** annotation
- @Test
  public void testConstructor()

Put it in a Test Class

- **Test class**: a Java class that contains unit tests.
- Naming convention: use the same name as the class you are testing and add "Test" to the end of the name. E.g. FooTest is a *test class* with unit tests for the class Foo
- Test classes should contain at least one test method.
- Test classes may also contain test fixtures

Setting Up Test Fixtures

- **Test fixture**: *initial conditions* that serves as the starting point for each test method in the test class.
- May be class variables or objects configured in any desired state best for running the tests.

Example

```
public class BankAccount {
        public double money = 0.00;
        public BankAccount(double amount) {
                money = amount;
        }
        public void deposit(double amount) {
                money += amount;
        }
        public double withdraw(double amount) {
                money -= amount;
                return amount;
        }
}
```

How do you tell if a test passes?

A Test Passes If…

- All its assertions pass
- "If the bar is green, the code is clean!"
- Means that all the behavior checked by the test was achieved

A Test Fails If…

- Any one of its assertions fails
- Means at least some part of the behavior checked by the test was not achieved

Suites

- **Test suite**: a collection of tests; one or more test classes
- Can run multiple test classes with a single test suite

Tradeoffs in Testing

- Simple → easy to write multiple, easy to test multiple
- Narrow → focus on a specific situation
- Similar → make sure same behavior for same actions
- Common → focus on expected situations
- Complex → longer to write, longer to test, more thorough
- Wide → cover a variety of situations
- Different → make sure same behavior for different actions
- Rare → ensure no errors in every situation

How are unit tests used?

- **Integration testing**: focuses on testing interconnected collections of units.
- **Functional testing**: focuses on ensuring an application meets its requirements.
- **Acceptance testing**: performed by a customer to assess whether the developer produced an acceptable product.
- **Incremental testing**: writing individual tests side-by-side with the corresponding code, writing a test for each feature or behavior as it is being developed.
- **Test-first coding**: writing the *tests* for each feature or behavior before writing the corresponding implementation.