Lab 7: Point/Circle/Shape classes

Introduction – READ THE INSTRUCTIONS FIRST!

The versions of class Point, Shape, and Circle given to you have some problems. It is your job to fix them so that they meet all the specifications in the comments of the source code and the test suites and the dialog below.

Be sure to keep a running log (Lab Notes) of each observed behavior, hypothesize what is wrong, how you fixed it, and how you tested the fix. Be sure to include erroneous hypotheses and non-fixes. Learning from mistakes is as important (if not more important).

These are mostly relatively minor 1 or 2 line programming fixes; do not redesign the classes.

Program Specifications

Make a new Lab7-Circle directory in your cs15u home directory and copy all the files from ~/../public/Lab7-Circle/ to your new Lab7-Circle directory.

There are five files: Point.java, Shape.java, Circle.java, TestPoint.java, and TestCircle.java. cd into your Lab7-Circle. Make another subdirectory named Original and copy all these files there.

    cd ~/Lab7-Circle
    mkdir Original; cp *.java Original

There is no program, per se. These are individual classes that are used among each other. The test programs are to perform unit tests of various constructors and methods within the classes.

The main motivation for this lab is to think about how and what to test in a class based on class specs. There are all sorts of Unit Testing frameworks (we will get to using JUnit soon). But they will not help you if you do not know what to test for.

Shape is an abstract class. Circle extends Shape. Circle has a Point that is its center. Point has an x and y coordinate.

Draw a simple UML diagram representing the relationship between Shape, Circle, and Point.

Follow these lab instructions:

We test Point first. TestPoint.java has the framework for testing class Point.

Go ahead and open Point.java and TestPoint.java in separate gvim edit sessions.

Looking at Point.java, we see a few ctors (no-arg, two-arg (int,int), and copy ctor), some accessor and mutator methods for the x and y coordinates, a move() method, toString(), equals(), and hashCode(). Someone was even kind enough to add a little logging framework in there for the equals() method.

Looking at TestPoint.java, we see the framework for performing unit tests on the ctors, the move() method, and the equals() method of class Point. The tests for the latter two are currently commented out. So we will focus on the unit tests for the various ctors in class Point first.
Compile and run the test program: TestPoint.java

```bash
$ javac TestPoint.java
$ java TestPoint
```

Good start. We actually Passed the first two ctor test cases, but failed the third one.

Edit TestPoint.java - look at testConstructors(). We are first testing the no-arg ctor. Look at how we are testing it. We log the specific type of test, create a new Point() invoking the no-arg ctor, then call a static test() method that takes two Strings - the expected String and the String that we got from what we are testing. Here we are using the toString() method from class Point to help us check our test results. So we had better make sure Point's toString() is implemented correctly. Just double-check that it is printing the name of the class followed by a colon and space and then the value of the x and y coordinates separated by a comma enclosed in parens. It looks okay.

So the no-arg ctor seems to be doing the right thing. And the second ctor test with the two-arg Point(int, int) ctor seems to be Passing also.

But the third ctor test passing in a negative x coordinate FAILED. The Specs for a Point in the Java graphics coordinate system state the values of the x and y coordinates should never be negative. We would expect a precondition check on the args of the Point(int, int) ctor to throw an IllegalArgumentException if either arg is negative. We are testing this ctor passing in a negative x coordinate. But is is Failing.

Time to look at this ctor in Point.java

Whoa! There are no precondition checks at all. Lucky for us, whoever wrote this class did the right thing and had the no-arg ctor invoke the two-arg ctor with the default values - so all the real work is focused in this one two-arg ctor. Look at how the no-arg ctor public Point() did this.

And also lucky for us, the two-arg ctor used the private mutator methods setX() and setY() to actually initialize the instance variables. This makes it nice and convenient for us to focus all of our precondition tests in the mutator methods setX() and setY().

So let's add an appropriate precondition check in each of the mutator methods. If the argument is negative, throw a new IllegalArgumentException with the message "setX(): " + x or "setY(): " + y.

Save, recompile TestPoint.java, test. Make sure the third ctor test Passes.

We have a test case for a negative x coordinate, but not for a negative y coordinate. Add a new test case in TestPoint.java for a negative y coordinate, say for new Point(25, -1). Model this test on the test for Point p3 with the same try-catch structure (let's use Point p4 for this fourth test) - we are looking for another IllegalArgumentException.

Save, recompile TestPoint.java, test.

You should now have four Passed tests under the Testing Constructors unit test.

There is another ctor we have not tested yet - the copy ctor - public Point(Point point). So let's add a test case for that ctor below the fourth test we just added. A copy ctor creates a new object as a copy of an existing object. Let's use p2 as the existing object. Write a test case modeled after the second ctor test case using the call

```java
Point p5 = new Point(p2);
```
as the middle line of this test. Get the right log.info() string and arguments to test() so this test case has the following output (with different time stamp, of course). Use p2.toString() as the expected arg to test().

Jul 19, 2012 6:33:00 PM TestPoint testConstructors
INFO: Testing: new Point(p2)
Jul 19, 2012 6:33:00 PM TestPoint test
INFO: Expected: "Point: (50,200)"
Jul 19, 2012 6:33:00 PM TestPoint test
INFO: Got: "Point: (50,200)"
Jul 19, 2012 6:33:00 PM TestPoint test
INFO: Passed.

Before going any further, can you think of any other value we should test with this copy ctor?

Hey! I said don't go any further until you think about it. Write what you think we should test _______________.

If you are having a hard time thinking about what you should test, ask one of the TAs. Note how a generic Exception is used to catch stray exceptions outside all of the test cases in this test method and FAIL.

Go ahead and write a test case for this. What should the ctor do in this case?

Again, this is an invalid argument (we really cannot create a new copy of a Point with this arg), so …

Recompile and test. Model this test case from one of the previous similar type of tests.

Once you get that test to Pass, we really should go back into Point.java and add a class invariant now that we know the x and y coordinates should never be negative. Similar to a previous lab, let's add a hasValidState() method at the end of Point.java. Fill in the appropriate body.

```java
private boolean hasValidState()
{
    // Check that this Point has a valid state: x and y are both >= 0.
    // Be sure to use the accessors getX() and getY() - do not access the
    // instance variables directly.
}
```

Then let's add a class invariant assertion check at the end of each ctor and method that can change the state of this Point: all the ctors and the two mutator methods and the move() method. You should have six assert statements:

```java
assert hasValidState() : "ctor Point() failed - not valid state.";
assert hasValidState() : "ctor Point(int, int) failed - not valid state.";
assert hasValidState() : "ctor Point(Point) failed - not valid state.";
assert hasValidState() : "setX() failed - not valid state.";
assert hasValidState() : "setY() failed - not valid state.";
assert hasValidState() : "move() failed - not valid state.";
```

as the last statement in the appropriate ctors, mutators, and move().

Recompile and run to make sure everything still works and Passes all the previous tests.

**SWITCH PAIR PROGRAMMING ROLES NOW!**
Now let us move on to the move() method. (Get it? Move on to the …)

The overall framework for writing unit tests for the move() method are already in place. The call to the testMove() method up in main() is commented out. Uncomment this call.

In the outer try block of testMove(), make a test case for testing a sample move(25, 25) - moving the x coordinate 25 pixels and the y coordinate 25 pixels. Here is an example:

```java
log.info("Testing: p1.move(25, 25)");
Point p1 = new Point();
p1.move(25, 25);
test("Point: (25,25)", p1.toString());
```

Understand what each line is doing before going further. Discuss with your partner. Looks like good test material.

Make sure this runs and Passes: Recompile TestPoint.java and java TestPoint

Let's test moving a Point such that the resulting x and/or y coordinate is negative. Model this test case on the above, say starting with a default (0,0) coordinate and move(-25,25) to make x negative.

```java
log.info("Testing: p2.move(-25, 25)");
Point p2 = new Point();
p2.move(-25, 25);
test("Point: (-25,25)", p2.toString());
```

Save, compile, and run with: java TestPoint

Hey! Why did it Pass???

Really - why did it Pass? Discuss with your partner. Looks like good test material.

First off, what were we really expecting? ____________________________

Rerun with the appropriate command line option. That is better. But we do not want to error out with an exception if someone erroneously (or maliciously) tries to move a Point such that the x or y coordinate becomes negative. We want to do something similar to the ctors where we use a precondition check to make sure the Point does not get into an invalid state.

But this time for the move() method, instead of throwing a new IllegalArgumentException (because the arguments may indeed be legal - we can have negative deltas to move the Point left and up), it is the overall operation of adding the deltas that results in a negative coordinate that is not allowed. So let's throw the UnsupportedOperationException to indicate that the requested operation (move to a negative coordinate) is not supported.

Add the appropriate precondition check at the beginning of the move() method and throw a new UnsupportedOperationException with the message "move(): would result in negative coordinate(s)"

In TestPoint, instead of expecting "Point: (-25,25)", we are expecting the UnsupportedOperationException - change the expected String in the call to test(). We also need to put this test case in a try-catch block to catch the UnsupportedOperationException. Model this similar to one of the test cases where we catch the IllegalArgumentException in one of the ctor test cases.
Save, compile, and run with: `java -ea TestPoint`

Once you get it working/Passing, add another test moving the y coordinate into the negative range similar to what you did with testing move() trying to change the x coordinate to a negative value.

Save, compile, and run with: `java -ea TestPoint`

Here is a good question - what would you expect if you were catching the `UnsupportedOperationException` in your test case, but move() was throwing another type of Exception?

What if we did not throw any exception from move() and did not have assertions enabled?

Here is what your last test case (to test a negative y value) should look something like:

```java
try {
    log.info("Testing: p3.move(25, -25)");
    log.info("Expecting UnsupportedOperationException");
    Point p3 = new Point();
    p3.move(25, -25);
    test("Expecting UnsupportedOperationException", p3.toString());
} catch (UnsupportedOperationException e) {
    log.info("Caught: " + e);
    log.info("Passed.
");
}
```

Notice the move() method is doing direct assignment into x and y. It should use the mutators instead. Change it so setX() and setY() are used instead.

Do we still need the precondition check in move()? Can we rely on the checks in setX() and setY() to catch the negative values? Try commenting out the precondition check in move(), compiling, and running again with assertions enabled. Why did only the first bad move(-25, 25) test run and not the second one move(25, -25)?

Which type of exception handling (precondition check for `UnsupportedOperation` or `IllegalArgumentException`) do you think is more appropriate for this move() method?

Uncomment the precondition check for `UnsupportedOperation` in move() and recompile and rerun the tests.

**SWITCH PAIR PROGRAMMING ROLES NOW!**

Now let us move on the to the equals() method.

The overall framework for writing unit tests for the equals() method are already in place. The call to the testEqual() method up in main() is commented out. Uncomment this call.

In the outer try block of testEqual(), make a test case for testing equals() passing in null. Just like String's equals() method, we should just return false right away if the argument to equals() is null. Here is an example:
log.info("Testing: p1.equals(null)");
Point p1 = new Point();
result = p1.equals(null);
test("false", "" + result);

Save, compile, run TestPoint.

The general try-catch block in testEqual() caught the NullPointerException and our test case FAILED. Better go make sure the equals() method is checking for an incoming null reference.

It is not. Add the check to return false if the parameter is null after the logging statements but before the instanceof check.

Save, compile, run TestPoint.

That is more like it! Now let's test passing an object in that is not a Point - say a String - to test the getClass() check for exact types. Remember using the Java operator instanceof in an expression like (o instanceof Point) checks that the object that the reference o references is related to (has an "is-a" relationship with) type Point. If we wanted to check for exact types, we use something like (o.getClass() == this.getClass()).

Model this test case similar to the one above testing null, but pass in a String like "Hello".

Yahoo! It Passed with false just like we were expecting. But I remember hearing/reading how one of the most common errors with **overriding** the equals() method from class Object is that most people erroneously end up **overloading** the equals() method instead of **overriding** the puppy.

What is the difference between method overriding and overloading? ________________________________

Luckily whoever wrote class Point put a logging statement in the equals() method. Yeah! We can use that to tell if we are really executing our equals() vs. class Object's equals(). Comment the line that is turning off logging.

Save, compiler, run TestPoint.

Drat! Our logging message "******** In Point's equals() ***********" did not print for the second of the two equals() tests. That means our equals() method is not really getting called (and not really getting tested!!!). What is the problem? If you need help, look at the signature of the equals() method as defined in class Object.

http://docs.oracle.com/javase/7/docs/api/java/lang/Object.html

To properly override a method, the method signature in the subclass must match exactly that in the superclass.

Fix the equals() method in Point so it really does override Object's equals().

Save, compile, run TestPoint.

Nice! Now let's add a couple more equals() test cases - one that should return true and one that should return false. Make sure they are separate objects, for example

    Point p3 = new Point(25, 25);
    Point p4 = new Point(25, 25);

and
Point p5 = new Point(5, 10);
Point p6 = new Point(10, 5);

Make sure these last two test cases Pass.

There is probably more testing we can do with class Point, but this is good for now for this lab. We have more lab goodness to experience.

A couple good Unix tools to know are diff and egrep.

We have not been using a source code repository system like subversion or git/github or sourceforge, etc. We have been focusing on testing and debugging more than software development. But one thing we would like to do is see the changes we made to the files. Remember I had you copy the original files to a subdirectory named Original? We can diff the original version of Point.java with our modified one to see the changes/additions we made.

$ diff Original/Point.java Point.java

A useful option is the -c option to give us some context: showing 3 lines above and below the differences. (See man diff)

$ diff -c Original/Point.java Point.java

Usually you specify the old/original file first and the new/updated file second.

As you noticed from running the TestPoint, there can be a lot of logging output. Redirecting output to a file to search (later) or preserve/document the testing can help.

Some standard bash shell redirection metacharacters:

- `<` redirect stdin
- `>` redirect stdout
- `2>` redirect stderr
- `&>` file redirect stdout & stderr to the same file
  (also: `>` file `2>&1` redirect stdout to file and then redirect stderr to the same)
- `2>&1 |` redirect stderr to stdout and pipe stdout to next command

$ java -ea TestPoint > TestPoint.output

Hmm … all of the logging output still came to the screen! What does this tell you about where logging is sent?

$ java -ea TestPoint &> TestPoint.output
or
$ java -ea TestPoint > TestPoint.output 2>&1

egrep is part of the grep family - $ man grep

$ egrep "FAIL" TestPoint.output
- should have none

$ egrep "Pass" TestPoint.output
Let's count the number of Passed tests:

```bash
$ egrep "Pass" TestPoint.output | wc -l
```
- pipe the output of egrep to wc (word count) -l (prints the # of lines). (See `man wc`)

### SWITCH PAIR PROGRAMMING ROLES NOW!

Now let us move on to some quick testing of class Circle. Look at `TestCircle.java`

The overall framework for writing unit tests for class Circle is already in place. Let's start with `testConstructors()`.

Class Circle extends Shape. Let's test Circle's no-arg ctor. The test case is already there for you in `TestCircle.java`

```bash
$ javac TestCircle.java
$ java TestCircle
```

It Failed! Take a close look at what we expected and what we got. What is different?

Edit Circle.java. Nice! Someone already populated this class with some logging, precondition checks in the mutator methods, and class invariants. Sweet!

But our first test case is Failing. Look at the no-arg ctor. It is just handing the initialization off to the `Circle(Point, int)` ctor. That ctor directly invokes the superclass ctor (Shape) passing in the name of this shape ("Circle"). Now look at the `toString()` method in Circle - it calls `getName()` to get the name of this Shape. There is no `getName()` method in class Circle, so we probably inherit it from class Shape. So let's look at `Shape.java`

Yep, there is `getName()` up in `Shape.java`. And the ctors for Shape end up calling `setName()` to set the name instance variable. Hmmm, why is the instance variable name in Shape null when we are passing the String "Circle" to the Shape ctor?

Hopefully by now you have seen this common programming error enough that you can spot the bug in `setName()`. Fix it. Save, compile, run TestCircle again. That's better!

We would want to test all the other ctors, but for now let's move on to test `getCenter()` accessor.

Uncomment the call to `testGetCenter()` in `main()` in `TestCircle.java`, save, compile, and run TestCircle.

The first test in `testGetCenter` (testing a deep copy) is Failing.

The comment above `getCenter()` in Circle.java says it should return a deep copy of the center Point. To test this we can call `getCenter()` to get the center Point of a Circle, and with each call, it should be returning a new Point that is a (deep) copy of the center Point (calling Point's copy ctor), not just a copy of the reference itself. Look at the `test()` line in the first test in `testGetCenter()` to test this. We want the references to be different.

Change the body of `getCenter()` to return a deep copy of the center Point (return a new Point object that is a copy of the existing center Point) vs. the actual reference as it is now. Make sure all the tests Pass.
Uncomment the call to testEquals() in main() in TestCircle.java. And we had better make sure our Circle-specific equals() is really getting called, so in Circle.java comment the log.setLevel(Level.OFF); line in the equals() method to turn on logging.

Compile and run. All test cases should run and Pass.

The version of equals() in Circle.java shows an example of using instanceof to check for a related type vs. getClass().

Run the tests and redirect stderr to stdout and then pipe stdout to egrep looking for "FAIL" in the input stream:

```
$ java TestPoint 2>&1 | egrep "FAIL"
$ java TestCircle 2>&1 | egrep "FAIL"
```

Count the number of Passing tests. Write the number displayed for each of the following:

```
$ java TestPoint 2>&1 | egrep "Pass" | wc -l  _____
$ java TestCircle 2>&1 | egrep "Pass" | wc -l  _____
```

Be sure to keep a running log (Lab Notes) of each observed bug, hypothesize what is wrong, how you fixed it, and how you tested the fix. Be sure to include erroneous hypotheses and non-fixes. Learning from mistakes is as important (if not more important).

Get one of the CSE 15L staff to check you off for the lab. You will need to go through your log describing each bug interaction and demo your fixed version with all three unit tests.