SENG275 – Lab 2
Due Monday Jan 31, 5:00 pm

Technical note: You now have a second repository on gitlab, named lab02

Welcome to Lab 2. Throughout this document, some sections will be in Red, and others will be in **bold**. The sections in Red will be marked. The questions in **bold** will not be graded. However, you should give these questions some thought, as these are the sorts of questions that could appear on a quiz or midterm.

Quick summary of what you need to do:

1. Install IntelliJ on the lab computers (unless you're going to use your laptop for this lab).
2. Read over the description of the Rectangle class you've been asked to create. Pay particular attention to the what objects of type Rectangle need to do, and try to notice elements of the design that have been left up to you.
3. Use a Test-Driven Development method to first design tests for the Rectangle class (in RectangleTest.java), then write your code for the class itself (in Rectangle.java). Continue the TDD cycle until the class and tests are complete.
4. Commit your changes to the lab03 repository, and push the commits back to gitlab.

   You're done!

Installing IntelliJ on the lab computers

We were unable to get the lab computers imaged with IntelliJ on them prior to the course beginning, so we need to install the IDE on these workstations. If you've already installed IntelliJ on your laptop and you're comfortable working on that platform, you can skip this step.

We'll be downloading IntelliJ from the Jetbrains website at jetbrains.com – click on the Download button in the upper right corner.

This will download `idealC-2021.3.1.tar.gz` into our Downloads directory. Move it to the directory where you'd like to install intelliJ – I suggest `~/intellij`.

Un-archive it with the following command:

```
tar xzvf idealC-2021.3.1.tar.gz
```

You can run intellij by entering the resulting `idea-IC-213.6461.79/bin` subdirectory and running the `./idea.sh` command.

Clone the lab02 project

The lab02 project has two files of interest to us – Rectangle.java, and RectangleTest.java. Other than the names, the structure of this project is the same as our code last week.

The Rectangle class

As you can see, the Rectangle class is nearly empty – we just have the bare-bones of a class present. One of our goals today will be to write this class.

We are deliberately NOT providing you with a specification for this class. Rectangles must have certain properties and be able to do certain things, but this will be described below in more general terms than you may be used to. This will force you to make certain design decisions as you go.

Given a general cartesian coordinate system with greater values toward the bottom and right, the following must be true of rectangles:

1. Their widths and heights must be positive integers greater than zero.
2. Rectangles have locations, which is the x,y coordinate of their upper-left corner.
3. We must be able to create a rectangle using new Rectangle(), which creates a rectangle whose upper left corner is at 0,0, and whose width and height are 1.
4. We must also be able to create a rectangle by specifying the x and y coordinates of the upper left corner, the width and the height at time of creation.
5. We must be able to fetch the x and y coordinates of the upper-left corner of the rectangle, as well as its width and height.
6. We must be able to change these values as well.
7. Any operation which could leave us with an invalid rectangle (one with a zero or negative width or height, must throw an exception.
8. We must be able to test rectangles for equality.
9. We must be able to ask a rectangle for its area.
10. We must be able to ask a rectangle if it contains another rectangle.

    Optional – higher difficulty:

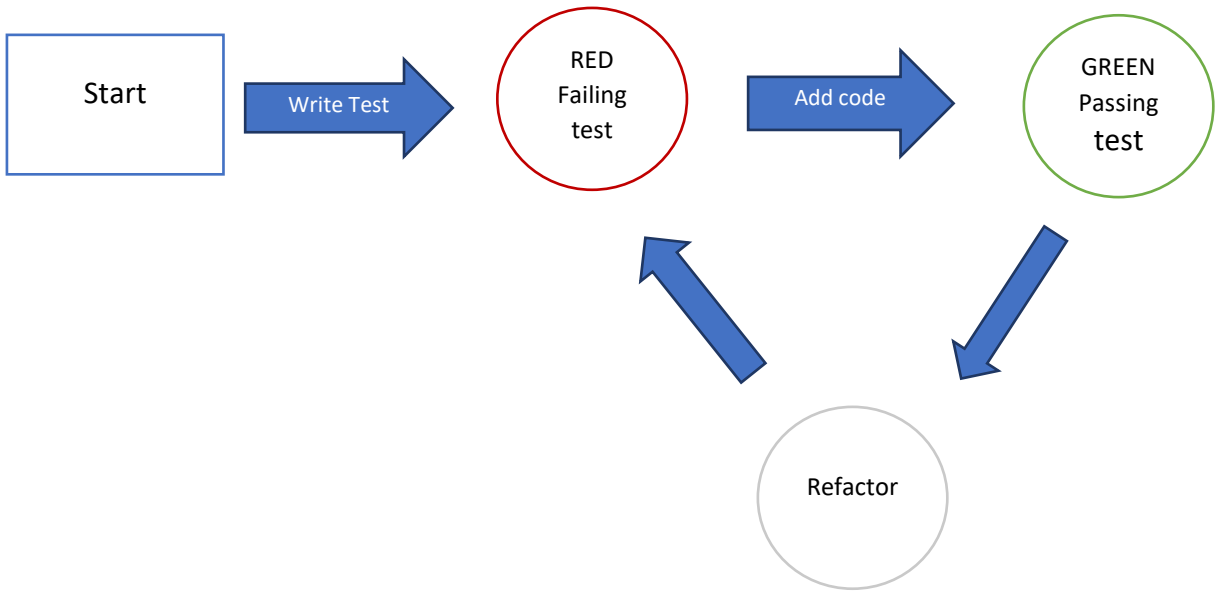
11. We must be able to *sort* rectangles from least to greatest area. Given an array of rectangles, Arrays.sort() should sort the array such that rectangles with lower areas come before rectangles with greater area.
12. We must be able to store and retrieve rectangles from a HashMap (so they must have valid hash values).

Note that some parts of these requirements are tightly-specified (obviously its area must be an integer, for example). But others are not – what does it mean for two rectangles to be *equal,* or for one rectangle to *contain* another? What should your methods be called? What arguments should they take, and what in what order? These are design decisions, and the tests you write will help you think about these sorts of issues.

Test-Driven Development

During the first lab, we wrote tests against a class that had already been written. Test-Driven Development inverts this order – we write tests first, and the process of trying to make the tests pass guides the development of our class. This method follows the Red-Green-Refactor procedure:

- RED: before we implement a feature, we write a test for that feature. Of course, this test should fail – it won't pass until we actually write the code. Make sure the test fails before continuing!
- GREEN: write the MINIMUM amount of code necessary to make the test pass. Resist the urge to skip ahead – as soon as the test passes, move on to…
- REFACTOR: Adding code may have introduced duplication, a lack of clarity, or 'magic numbers' that could be replaced by named constants. Address these purely stylistic issues, then resume with the RED step.



Guided Example

The rectangle class is nearly empty – we have no constructors, and just enough code to allow us to instantiate a default rectangle, with x and y at zero, and width and height at one. Our test code is simple as well; we have one 'test' that just returns true no matter what (so we know our framework is working) and one test that checks the default rectangle.

Now it's time to start TDD. The first step is to write a failing test. Checking the list of requirements, we're told that we should be able to create a rectangle by specifying the location, width and length to a constructor. Let's try a test that does that. In rectangletest:

```
@Test
void customRectangle() {
   Rectangle r = new Rectangle(5, 5, 10, 10);
   assertEquals(5, r.getX());
   assertEquals(5, r.getY());
   assertEquals(10, r.getWidth());
   assertEquals(10, r.getHeight());
}
```

This won't even compile – it's not a real RED test unless it compiles. So we're still at the RED stage – add enough code to allow this to compile. Add to rectangle.java:

```
public Rectangle() {}
public Rectangle(int x, int y, int width, int height) {}
```

Now the test fails (we get a yellow X for customRectangle) and we have finished the RED stage of TDD. Now on to the GREEN stage – we modify rectangle.java as necessary to make the failed test pass, WITHOUT causing any of the other tests to fail (so we can't just hard-code new values into the accessor functions – defaultRectangle would fail if we did).

We have to give up hardcoding and create some private member variables. Let's do that now. In rectangle.java:

```
private int x;
private int y;
private int width;
private int height;

public Rectangle() {
   x=0; y=0; width=1; height=1;
}
public Rectangle(int x, int y, int width, int height) {
   this.x=x; this.y=y; this.width=width; this.height=height;
}
public int getX() { return x; }
public int getY() { return y; }
public int getWidth() { return width; }
public int getHeight() { return height; }
```

I've compressed the code a little to save space on the page; you should use the coding conventions you're most comfortable with.

All the tests pass – we've now completed the GREEN stage. Now it's time to REFACTOR. Sometimes we'll skip this step, sometimes we won't. Skim over the code to see if there's anything obviously redundant, awkward or confusing about it.

In my case, I don't really like the default constructor – I have a custom constructor, so why not chain them together? In the REFACTOR stage, I rewrite the default constructor:

```
public Rectangle() {
   this(0,0,1,1);
}
```

I think that's nicer – you may disagree! Part of the fun of TDD is using the process to try things out, and everyone's going to make different choices as we go along. I don't see any more refactoring opportunities, so time for a RED test again:

Let's try a setter function: in rectangletest.java:

```
@Test
void changeWidthValid() {
   Rectangle r = new Rectangle(5,5,10,10);
   r.setWidth(20);
   assertEquals(20, r.getWidth());
}
```

And in rectangle.java (to allow it to compile):

```
public void setWidth(int width) {}
```

It would be easy to fill out the body of the function (we know how to write setter methods) but resist the urge to do so. Part of TDD is about using the process to stumble upon unexpected complications in your code during testing, so they don't bite you during execution.

Only once we get the RED test result should we go ahead and modify the method:

```
public void setWidth(int width) {
   this.width = width;
}
```

How closely you stick to this pattern will depend a little on how helpful you find it – as you get more practice with TDD you may find yourself bending the rules a little, which is fine – TDD is just one more tool among many in your toolkit.

No need to REFACTOR this time, so we'll go straight back to RED. Dealing with Exceptions requires a little different syntax, so we'll try one more step:

```
@Test
void changeWidthNegative() {
   Rectangle r = new Rectangle(5, 5, 10, 10);
   r.setWidth(-10);
   assertThrows(IllegalArgumentException.class, () -> {r.setWidth(-10);});
}
```

Here we're asserting that the method setWidth should throw an exception if we try to set the width to a negative value. We need to use a lambda expression as shown – this is a requirement of Junit.

Since getWidth() does not throw an exception yet, this test is RED. With that stage completed, we can move on to GREEN, and continue with our TDD lab!

Rules for this exercise

The goal of this lab is to give you a chance to experience the TDD method and see if you enjoy it. Some people find it repetative and irritatingly slow. Others love the rhythm of the TDD cycle and the reward of seeing red tests turn green. The only way to find out is to try it.

Accordingly, we require that you commit and push your code at least 3 separate times while developing your solution. Do so once after you've got the code from the guided example above, and then at least twice more. Each time should be at the point you have a RED test.

Hints

Part of this exercise is intended to stretch your java muscles a little. It may take a little practice to get back to familiarity with the language, and you may have to look some things up. The following hints will hopefully help you:

- You'll be marked based on whether you tried out the TDD method, and can prove it through your gitlab commit log. It's better to try several cycles of TDD, committing after each, than to rush through coding everything as soon as you can. This will be our only TDD lab, and it may be your only opportunity to try out this controversial and interesting approach to coding.
- Tests for thrown exceptions have unusual syntax – consult the guided example above.

- We have deliberately not specified names of methods, how many you need, how many tests are required, what the exact types of exceptions you need to throw, etc. Experiment and come up with your own design. If it meets the requirements you'll get full marks.
- Testing an object for equality involves overriding the equals method. Remember the distinction between two object references that are the Same (x == y) and two objects that are Equal (x.equals(y)). Requirement 8 concerns Equality.
- Remember that an invalid Rectangle is one with a negative width or height. ANY method that might put a rectangle into an invalid state is one that needs to throw an exception.
- The optional requirements are… optional. Don't sweat them, they're not for extra marks. If you take them on, you'll be doing more coding. You'll need to override more methods, and learn how objects are compared and hashed. For sort, try creating an array of Rectangles and calling Arrays.sort on them, and then assert that the resulting array is in the order you expect. You'll need to tell Java how to compare two Rectangles. For HashMap, you'll need to make sure that your Rectangle object has a unique (or reasonably unique) hash value.

Reference: Junit 5

The best way to explore the Junit assertions is to just start typing assert in your code and see what's out there. For clarity, though, here's a list of some of the most commonly used assertions:

assertEquals(*expected, actual*)
assertNotEquals(*expected, actual*)
assertSame(*object reference, object reference*)
assertNotSame(*object reference, object reference*)
assertArrayEquals(*expected_array, actual_array*)
assertThrows(*exception.class, lambda expression*)
assertDoesNotThrow(*exception.class, lambda expression*)
assertTrue(*boolean expression*)
assertFalse(*boolean expression*)
assertStringsMatch(*List<String>, List<String>*)
assertNull(*object reference*)
assertNotNull(*object reference*)