Homework 2: Classes
Due: 5:00 PM, February 17, 2017

Contents

1 Students and Courses (Practice) 3

2 Posn (Practice) 3

3 Rational Numbers 6

4 Shape-ology 7
   4.1 Rectangle: Take 1 8
   4.2 Triangle 9
   4.3 Square: Take 1 9
   4.4 Quadrilaterals 10
   4.5 Rectangle and Square: Take 2 11

5 Appendix 12

Objectives

By the end of this homework, you will know:

- all about object-oriented programming

By the end of this homework, you will be able to:

- write toString and equals methods

How to Hand In

For this (and all) homework assignments, you should hand in answers for all the non-practice questions. For this homework specifically, this entails answering the Rational Numbers question and the Shape-ology question.

In order to hand in your solutions to these problems, they must be stored in appropriately-named files with the appropriate package header in an appropriately-named directory.

Begin by copying all source code (/course/cs018/src/hw02/src/*.java) to your ~/course/cs018/workspace/javaproject/src/hw02/src directory. After doing so, the following source code
files should be in your `~/course/cs01800/workspace/javaproject/src/hw02/src` directory, all as part of the `hw02.src` package:

- `IRational.java`, exactly as given.
- `ZeroDenominatorException.java`, exactly as given.
- `IShape.java`, exactly as given.
- `Posn.java`, exactly as given.
- `Line.java`, exactly as given.

After completing this assignment, the following source code files should be in your `~/course/cs01800/workspace/javaproject/sol/hw02/sol` directory, all as part of the `hw02.sol` package:

- **Rational Numbers**
  - `Rational.java` containing `public class Rational`, which implements `IRational`.
  - `RationalTest.java` containing `public class RationalTest`.

- **Shape-ology**
  - `BasicRectangle.java` containing `public class BasicRectangle`, which implements `IShape`.
  - `BasicRectangleTest.java` containing `public class BasicRectangleTest`.
  - `Triangle.java` containing `public class Triangle`, which implements `IShape`.
  - `TriangleTest.java` containing `public class TriangleTest`.
  - `BasicSquare.java` containing `public class BasicSquare`, which extends `BasicRectangle`.
  - `BasicSquareTest.java` containing `public class BasicSquareTest`.
  - `Quadrilateral.java` containing `public abstract class Quadrilateral`, which implements `IShape`.
  - `Parallelogram.java` containing `public class Parallelogram`, which extends `Quadrilateral`.
  - `ParallelogramTest.java` containing `public class ParallelogramTest`.
  - `Trapezoid.java` containing `public class Trapezoid`, which extends `Quadrilateral`.
  - `TrapezoidTest.java` containing `public class TrapezoidTest`.
  - `Rectangle.java` containing `public class Rectangle`, which extends `Parallelogram`.
  - `RectangleTest.java` containing `public class RectangleTest`.
  - `Square.java` containing `public class Square`, which extends `Rectangle`.
  - `SquareTest.java` containing `public class SquareTest`.

To hand in your files, navigate to the `~/course/cs01800/workspace/javaproject/` directory, and run the command `cs018_handin hw02`. This will automatically hand all of above files. Once you have handed in your homework, you should receive an email, more or less immediately, confirming that fact. If you don’t receive this email, try handing in again, or ask the TAs what went wrong.
Practice

1 Students and Courses (Practice)

Task: Write (and test) Student and Course classes. The Student class should include as attributes things like name, date of birth, home state, and concentration. The Course class should include as attributes things like title, department, number, professor, and an indicator of whether the class is Mandatory S/NC.

Note: Don’t forget to write toString and equals methods for both your classes.

Task: Now that you have written these two classes, test your code by creating a few students and a few courses.

Task: Practice composing these two classes. For example, you can add a FavoriteCourse attribute, which is-a Course, to Student. Likewise, students can be attributes of a course. Be sure to add additional corresponding tests to your code, after implementing these extensions.

2 Posn (Practice)

Note: The solution to this practice problem is provided in this week’s support code, as support for one of the problems on this homework. We recommend you have a go at solving this problem yourself, before reading the source code provided.

Let’s think way back to the (glory) days of CS 17, when we created structs in Racket. Specifically, we created a struct called posn, which first allowed us to describe points, and later lines and shapes.

In Java, a simple Posn class could be written as follows:

```java
/**
 * A class that represents points in the Cartesian plane, given their x and y coordinates.
 */
public class Posn {

    public double x;
    public double y;

}
```

You might also remember that in Racket, to construct a Posn, we would use the Racket-generated function make-posn. In Java, and in other OOP languages, when creating a class, we write the code for the class’ constructor inside the class itself, like this:

```java
public class Posn {

    public double x;
    public double y;

```
/**
* @param x - x coordinate
* @param y - y coordinate
*/
public Posn(double x, double y) {
    this.x = x;
    this.y = y;
}

This is a very simple constructor. It merely assigns x and y the values of their respective inputs.

Here are two examples of this new class in use:

    Posn p1 = new Posn(17.0, 18.0);
    Posn p2 = new Posn(-15.0, -16.0);

**Task:** Create a Posn class and a constructor. Create a main method and make Posns p1 and p2. Try printing out p1 and p2. Is that what you expected?

What do we want Java to print out when we ask for the value of p1? I’d venture to say we want something like (17.0, 18.0). Luckily, Java has a way for us to create this output, or anything similar that we might prefer.

Every class has a method called toString, because every class inherits from the Java class Object, which sits at the root of the Java class hierarchy, and Object has a toString method. This toString method is called by System.out.println whenever its argument is an object. The default behavior of this method, which is inherited from Object, is to print out the object’s memory location. As that’s rarely of any use, in CS 18, we write our own toString method for all our classes. But since every class already has a toString method, we must override the inherited definition. We do this using the @Override annotation[1].

**Task:** Write a toString method for Posn.

Similarly, every class has a method called equals, which it also inherits from Object.

The default behavior of equals is to use == to check equality. But == checks if the two objects are equal. That is, it checks if the two objects are located in the same place in memory. Sometimes this is useful, but most of the time you really want to know if the two objects represent the same thing, meaning their attribute values are equal. This is where equals comes in. By overriding the default equals method with your own, you check whether the two objects represent the same thing.

**Task:** Write an equals method for Posn.

**Task:** Create a tester class for Posn called PosnTest. Write a few tests in in this class to confirm that your .equals method works as intended in your Posn class.

---

[1] Actually, the @Override annotation is not strictly necessary for a Java program to compile or run. However, it is good style to use it, so in CS 18 it is strictly necessary. That is, we require that you use the @Override annotation whenever you are overriding a method.
Our Posn design at present has some positives (e.g., toString and equals), but it also has some negatives. Most notably, the attributes are all public! This means that they can be accessed by code that creates myPosn and then invokes myPosn.x and myPosn.y outside the Posn class. This is not good coding practice!

Consider, for example, a class with two attributes, one an array and a second the index into the array. If the index is public, then it could accidentally be set to a negative value! Using a setter, which is a public method that is used to set attribute values, one could enforce an integrity constraint, namely, that the index cannot be negative, before agreeing to set the attribute to the new value.

```java
public void setIndex(int i) {
    if (i < 0) {
        System.out.println("Sorry, but that is not a valid index.");
    } else {
        this.myIndex = i;
    }
}
```

So, to improve our design, we will now make x and y private. In addition, we will write public getter and setter methods that allow other classes to access and modify the variables safely. In this example, the getter and setter methods aren’t doing all that much; but it is nonetheless good practice to structure your code this way so that it is easy to add integrity constraints as the need arises.

The new implementation looks like this:

```java
public class Posn {

    private double x;
    private double y;

    public Posn(double x, double y) {
        this.x = x;
        this.y = y;
    }

    /**
     * @return the x coordinate
     */
    public double getX() {
        return this.x;
    }

    /**
     * @return the y coordinate
     */
    public double getY() {
        return this.y;
    }

```
public void setX(double x) {
    this.x = x;
}

public void setY(double y) {
    this.y = y;
}

Task: Modify your Posn class so that it uses getters and setters. Test these methods in your PosnTest class.

Task: Once you have completed this practice problem, spend some time comparing your solutions with the Posn and PosnTest classes provided in the src directory.

Problems

3 Rational Numbers

A rational number is a fraction in which both the numerator and denominator are integers. Among other operations, rational numbers can be added and multiplied together. In this problem, you will write a class that represents rational numbers by implementing an interface that allows you to do simple arithmetic like this with them.

Task: We have provided an IRational interface that includes four methods: getNumerator, getDenominator, plus, and times. Begin by copying /course/cs0180/src/hw02/src/IRational.java to your ~/course/cs0180/workspace/javaproject/src/hw02/src directory.

Task: Take a look at the IRational interface. You will notice that some of the methods throw a ZeroDenominatorException. We have written a ZeroDenominatorException class for you, and we’ll explain how to you should use it a little later in the problem. For now, you should copy /course/cs0180/src/hw02/src/ZeroDenominatorException.java to your ~/course/cs0180/workspace/javaproject/src/hw02/src directory.

Note: Any of your solution files which reference this interface or class must import them like this:

```java
import hw02.src.IRational;
import hw02.src.ZeroDenominatorException;
```

Task: Create a Rational class whose constructor takes as input two integers, a numerator (the
first) and a denominator (the second). Be sure that your class implements the \texttt{IRational} interface. Unfortunately, this simple design allows us to represent rationals that do not make sense. What number does \texttt{new Rational}(1,0) represent? We want to avoid this division by zero.

\textbf{Task}: Modify your constructor so that it throws a \texttt{ZeroDenominatorException} exception if the denominator is zero.

This design is still too simple because it allows too many representations of each rational number. For example, the rational number 2 can be represented by \texttt{new Rational}(2, 1), \texttt{new Rational}(4, 2), and \texttt{new Rational}(100, 50).

\textbf{Task}: Modify your code so that rational numbers are represented in lowest terms. That is, that the greatest common divisor (gcd) of the numerator and denominator must be 1.

\textbf{Hint}: To find the GCD of two numbers you can use Euclid’s method:

$$gcd(a, b) = \begin{cases} a & \text{if } b = 0 \\ gcd(b, a \mod b) & \text{otherwise} \end{cases}$$

\textbf{Note}: This method of finding the GCD is not the same one you implemented in CS 17, as this method works with both postive and negative (non-zero) integers.

\textbf{Task}: Write the methods \texttt{plus} and \texttt{times}. The \texttt{plus} method takes as input an \texttt{IRational} and adds it to this \texttt{Rational}, returning the result as a new \texttt{Rational}. The \texttt{times} method behaves analogously, except that it performs multiplication.

\textbf{Hint}: Since each of these methods creates a new \texttt{Rational} object, they could potentially throw a \texttt{ZeroDenominatorException}, so you must write your method declarations accordingly. Refer to the \texttt{IRational} interface to see how these declarations should look.

\textbf{Task}: Write a \texttt{toString} method for the \texttt{Rational} class.

\textbf{Task}: Override the \texttt{equals} method to test for deep equality.

\textbf{Hint}: Don’t forget that \texttt{equals} takes as input an argument of type \texttt{Object}!

\textbf{Note}: Throughout, don’t forget the \texttt{Override} annotation!

4 \hspace{1em} \textbf{Shape-ology}

Townsville, USA has been peaceful lately, so the three Powerpuff Girls have been tasked with helping local schoolkids learn their shapes! Unfortunately, the trio doesn’t use geometry much, so they’ve forgotten a lot of the properties of different shapes. Help our heroes by creating a few shape classes that teach some basic geometry.

The shapes you will be implementing, \texttt{Rectangle}, \texttt{Triangle}, and \texttt{Square}, make use of \texttt{Posn} and \texttt{Line} classes, which are very similar to the \texttt{posn} and \texttt{line} structures of CS 17. Java versions of these classes are provided for you in the \texttt{/course/cs0180/src/hw02/src/} directory. Begin by copying \texttt{/course/cs0180/src/hw02/src/Line.java} and \texttt{/course/cs0180/src/hw02/src/Posn.java} to your \texttt{-/course/cs0180/workspace/javaproject/src/hw02/src} directory. Any of your solution files which reference these classes must import them like this:
import hw02.src.Line;
import hw02.src.Posn;

Before starting on their star charts, Blossom, Bubbles, and Buttercup first sit down to discuss the fundamental nature of shapes. They rightly conclude that rectangles, squares, triangles, parallelograms, and trapezoids are all shapes; as such, they have certain attributes in common. Simply put, the area and perimeter of each shape can be calculated, albeit in slightly different ways. Java allows us to abstract out this shared quality as an interface.

Task: We have provided an IShape interface that includes two methods: getArea and getPerimeter. Copy /course/cs0180/src/hw02/src/IShape.java to your ~/course/cs0180/workspace/javaproject/src/hw02/src directory.

As you proceed to work on this problem, make sure that all your classes implement this interface. This is a way of creating requirements for yourself, which is always good programming practice.

4.1 Rectangle: Take 1

After pestering Buttercup one too many times about the geometric qualities of rectangles, Bubbles finds herself on her own and unsure of how to determine the various properties of a rectangle. It is up to you to help her implement methods that do so.

Task: The rectangle class implemented in lecture was characterized by its width and height, but a rectangle can also be described as four Lines. Write a BasicRectangle class, whose internal representation of rectangles uses four lines, and whose constructor looks like this:

```java
public BasicRectangle(Posn bottomLeft, double width, double height) {
    // your code here
}
```

Note: You may assume that the values of width and height are positive.

Your BasicRectangle class should implement the IShape interface. As such, it must include the method getArea(), which returns the area of the rectangle, and the method getPerimeter(), which returns the perimeter of the rectangle.

Hint: Your object’s area and perimeter may be requested multiple times, but those values only need to be calculated once. Make sure to cache your data accordingly.

Hint: Make sure to give your variables the appropriate permissions. Start by making everything private with getters/setters as necessary. Change the permission to protected if it needs to be used by subclasses.

Given an instance myRectangle of your BasicRectangle class, wouldn’t it be nice to be able to print myRectangle using System.out.println(myRectangle)? For this statement to print something intelligible (i.e., something other than a memory address), you need to specify how a BasicRectangle should be printed. You can do this by overriding the toString method that your class inherits from Object.

Task: Add a simple toString method to your BasicRectangle class, which returns a string identifying your shape as a rectangle, as well as the particulars that characterize each rectangle.
Note: Almost all classes you ever write (and certainly all the classes for this problem) should have their own `toString` method.

It would also be nice to check whether two instances of `BasicRectangle` are equal. However, the default `equals` method inherited from `Object` checks only for shallow equality.

Task: Override the `equals` method to check for deep equality.

Note: Like `toString`, almost all classes you ever write (and certainly all the classes for this problem) should have their own `equals` method.

Hint: We provide sample `toString` and `equals` methods for you in the `Line` and `Posn` classes. Follow our model in the corresponding methods you write for your classes.

### 4.2 Triangle

Now that Bubbles has rectangles all figured out, your next job is to write a class for her that represents a triangle.

Task: Write a `Triangle` class that implements `IShape`. Be sure to include `toString` and `equals` methods with appropriate behavior.

Important: The `Triangle` constructor should take as input three `Posns`, but it should then use three `Lines` to represent a triangle.

Hint: You may assume that the three `Posns` input to your constructor are all distinct so that they form a geometrically valid triangle.

Note: If \( a = \text{new Posn}(0, 0) \), \( b = \text{new Posn}(0, 1) \), and \( c = \text{new Posn}(1, 0) \), then
\[
\text{new Triangle}(a, b, c), \text{new Triangle}(a, c, b), \text{new Triangle}(b, a, c), \\
\text{new Triangle}(b, c, a), \text{new Triangle}(c, a, b), \text{and} \text{new Triangle}(c, b, a)
\]
all describe the same triangles. Be sure that your `Triangle` class you implement accurately reflects these identities!

Another Hint: Recall Heron’s formula for computing the area \( A \) of a triangle:

\[
A = \sqrt{s(s-x)(s-y)(s-z)}
\]

Here \( x, y, \) and \( z \) are the side lengths of the triangle, and \( s \) is half its perimeter.

### 4.3 Square: Take 1

Whenever one concept is a special case of another, we can define the former as a subclass (i.e., subtype) of the latter. By so doing, the subclass can inherit all the data and functionality of the superclass. This way, you can implement additional features in the subclass, but do not need to rewrite any features that it inherits from its superclass.

Task: Write a `BasicSquare` class (including `toString` and `equals` methods), whose constructor looks like:

```java
public BasicSquare(Posn bottomLeft, double sideLength) {
    // your code here
```
Write your constructor in as few lines as possible (i.e., only 1, in addition to the header).

**Hint:** A square is a special case of a rectangle: use inheritance!

**Note:** You may assume that the value of `sideLength` is positive.

### 4.4 Quadrilaterals

Your next task is to create two more classes, `Parallelogram` and `Trapezoid`, which also implement the `IShape` interface. Because Bubbles’ brain is pretty fried by now, these classes need only to represent shapes with two sides parallel to the x-axis. Moreover, you may assume that any inputs to your constructors create valid parallelograms and trapezoids (i.e., they always have non-negligible areas and perimeters).

When writing these two new classes, you may find yourself copying a lot of code from one to another. Whenever you find yourself reimplementing the same data and/or functionality in more than one place, it’s a good idea to step back and re-think your design. More specifically, you should consider abstracting out common data and functionality into a single place. Often, this requires changing your original design, but the end result is usually cleaner, easier to read, and most importantly, much easier to maintain.

What do parallelograms and trapezoids have in common? Like rectangles, triangles, and squares, they share area and perimeter calculators. But they share so much more, because they are both quadrilaterals! Consequently, their data representations may be similar.

**Task:** Write a `Quadrilateral` abstract class which implements the `IShape` interface.

**Hint:** Think about what attributes and behaviors all `Quadrilaterals` share. Do you need this full generality to implement parallelograms and trapezoids? Figure out exactly what you do need, and then implement an abstract `Quadrilateral` class.

**Task:** Write a `Parallelogram` class whose constructor looks like:

```java
public Parallelogram(Posn bottomLeft, Posn topLeft, double bottomLength) {
  // your code here
}
```

**Task:** Write a `Trapezoid` class whose constructor looks like:

```java
public Trapezoid(Posn bottomLeft, Posn topLeft, double bottomBaseLength, double topBaseLength) {
  // your code here
}
```

**Hint:** Can your `Parallelogram` and `Trapezoid` classes share code? If so, they should both be fairly short, because they can extend a common class—`Quadrilateral`. 
4.5 Rectangle and Square: Take 2

Just as a square is a special case of a rectangle, a rectangle is a special case of a ... what?

Task: Write new Rectangle and Square classes that fit into your new, expanded class structure with the following constructors, which mimic those of your BasicRectangle and BasicSquare:

```java
public Rectangle(Posn bottomLeft, double width, double height) {
    // your code here
}

public Square(Posn bottomLeft, double sidelength) {
    // your code here
}
```

Note: Be sure not to modify your BasicRectangle and BasicSquare classes! You need to hand in both versions.
5 Appendix

Here is a class diagram that depicts the structure that the source files and your solution files should have after you finish implementing Shapes:
Please let us know if you find any mistakes, inconsistencies, or confusing language in this or any other CS18 document by filling out the anonymous feedback form:

http://cs.brown.edu/courses/cs018/feedback