LAB 3 (part A)

Create a project named **`lab3`** for your work. Don't use the default package (rather, create at least one to contain your work). Framework code and this lab are available on moodle.

Lab 3 is due on Friday, March 27th, before the end of the day. But, get it done early so you don't get your spring break off on the wrong foot!

1. Aliens redux

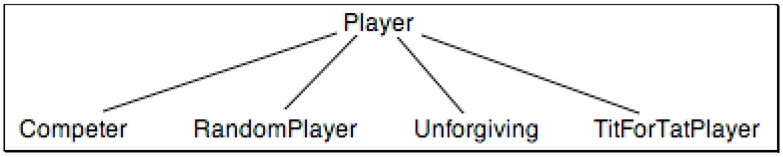
This problem is taken from the book: programming project 8.1 on page 518 (in the 5th edition), and is reproduced in the shaded box below.

> In Programming Project 7.3 from Chapter 7, the **`Alien`** class was rewritten to use inheritance. The rewritten **`Alien`** class should be made **`abstract`** because there will never be a need to create an instance of it, only its derived classes. Change this to an **`abstract`** class and also make the **`getDamage`** method an **`abstract`** method.
>
> Rerun your tests from your **`AlienTester`** class to ensure that it still operates as expected.

Copy your package and classes in your solution to the Aliens problem in lab 2 into your lab 3 project. Additionally, convince yourself that you cannot make an instance of the **`Alien`** class.

2. The prisoner's dilemma redux

Recall the prisoner's dilemma problem from lab 2. Your class hierarchy could be represented as:



There is a flaw in this organization: the **`Player`** class is a player on its own, one that always cooperates. Thus the "is-a" relation between the other player classes and `Player` doesn't quite hold. What the `Player` class should provide is a sort of placeholder for the **`cooperates`** method; an inheriting class would then provide the details.

Copy your solution to the prisoner's dilemma problem in lab 2 into your lab 3 project. Refactor your class hierarchy to fix this flaw, and include a detailed comment in your tester class explaining your new class hierarchy and how it solves the problem from above. You'll probably want to use abstract classes and/or methods.

Run some of your tests from lab 2 to ensure that your code still works, and include the output in a comment.

3. Simple shape hierarchy

Write an inheritance hierarchy for classes of simple shapes.

- Create a class **Shape.** Derive the four classes **ZeroDimensionalShape**, **OneDimensionalShape, TwoDimensionalShape** and **ThreeDimensionalShape** from the **Shape** class. Derive the classes **Point**, **Line**, **Circle**, and **Sphere** from these four classes.
- Use abstract classes to define **Shape**, **OneDimensionalShape**, **TwoDimensionalShape** and **ThreeDimensionalShape** classes, and then implement them in the **Line**, **Circle** and **Sphere** classes.
- Include a **shapeID** variable in the **Shape** class and a **getID** method to get the ID for each shape created. Create a unique ID for each instance of a Shape. Include an abstract **move()** method to move the shape in the x, y, and z directions.
- Derive the **Point** class from **ZeroDimensionalShape**. Give the point class an X, Y, and Z coordinate, and provide methods to get and set the coordinates.
- Derive the **Line** class from **OneDimensionalShape**. One dimensional shapes have an abstract **getLength()** method. The line class is constructed using two points, example: **new Line(new Point(0,0,0),new Point(4,4,4))** creates a line from location 0,0,0 to location 4,4,4.
- Derive the **Circle** class from **TwoDimensionalShape**. Two dimensional shapes have an abstract **getArea()** method. The circle class is constructed using a point for the center, and a radius: e.g., **new Circle(new Point(2,2,2),2)** creates a circle centered at 2,2,2 with radius of 2. Assume the circle has the same Z value for all points in the circle.
- Derive the **Sphere** class from **ThreeDimensionalShape**. Three dimensional shapes have both an abstract **getArea()** and abstract **getVolume()** methods. The sphere class is constructed using a point for the center, and a radius, example: **new Sphere(new Point(2,2,2), 2)** creates a sphere centered at 2,2,2 with a radius of 2.
- Include a **toString()** that returns a **String** description of the key properties of each object, including its ID.
- Use a **ShapeTester** class to test the **Shape** hierarchy. Illustrate the use of <u>polymorphism</u> in your tester. For example, first create an **ArrayList** of **Shape**s. Using loops, print the current location, move the shapes, and print the new location. For instance:

```
// possible shape testing code
for (int i = 0; i <  shapes.length; i++) {
      if (shapes.get(i) instanceof OneDimensionalShape) {
          OneDimensionalShape ods = (OneDimensionalShape)shapes.get(i);
          System.out.printf("%s length is %f\n",ods,ods.getLength());
      }
      if (shapes.get(i) instanceof TwoDimensionalShape) {
          TwoDimensionalShape tds = (TwoDimensionalShape) shapes.get(i);
          System.out.printf("%s area is %f\n", tds, tds.getArea());
      }
      if (shapes.get(i) instanceof ThreeDimensionalShape) {
          ThreeDimensionalShape tds = (ThreeDimensionalShape) shapes.get(i);
          System.out.printf("%s area is %f\n", tds, tds.getArea());
          System.out.printf("%s volume is %f\n", tds, tds.getArea());
      }
  }
```