CS 1130, LAB 4: ABSTRACT CLASSES

Name: _____ **Net-ID**: _____

There is an online version of these instructions at

http://www.cs.cornell.edu/courses/cs1130

You may wish to use that version of the instructions.

This lab introduces you to the useful software-engineering concepts of an "abstract class" and an "abstract method". The topic is covered in Section 4.7 of the class text and on lesson page 4-5 of the ProgramLive CD. We summarize the material below (in Section **??**, **Introduction to Abstract Classes**), but your lab instructor will present the concepts to you (which are quite simple) if you ask.

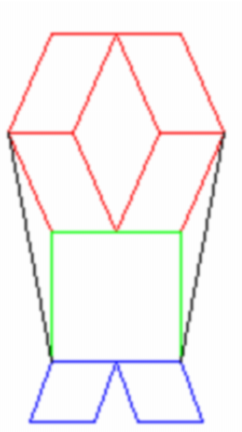
This lab also gives you some more practice in reading Java programs and, in the context of this lab, modifying them to draw some shapes in a JFrame.

Requirements For This Lab. There are several files necesssary for this lab, and they are all available from the online version of these instructions at the course web page. You should create a new directory on your hard drive and download the following five files into this directory:

- `DemoShapes` (http://www.cs.cornell.edu/courses/cs1130/2012fa/labs/lab4/DemoShapes.java)
- `Shape` (http://www.cs.cornell.edu/courses/cs1130/2012fa/labs/lab4/Shape.java)
- `Parallelogram` (http://www.cs.cornell.edu/courses/cs1130/2012fa/labs/lab4/Parallelogram.java)
- `Rhombus` (http://www.cs.cornell.edu/courses/cs1130/2012fa/labs/lab4/Rhombus.java)
- `Square` (http://www.cs.cornell.edu/courses/cs1130/2012fa/labs/lab4/Square.java)

To use these files, open them all up in DrJava and compile them. In the Interactions Pane, create an instance of DemoShape with the command:

```
DemoShapes d = new DemoShapes();
```



A figure like that on the right should appear in the window. The output in the Java console describes seven shapes that are drawn in the window.

This lab will be a combination of written exercises and modifying the files above. As in the last lab, there is no JUnit test because you will be able to test al of your code visually with `DemoShapes`. Furthemore, to simplify the lab submission, we are not asking you to show off your .java files. Instead, we have given you space in this lab to write down what you have done on paper (starting in the Section **??**, **Lab Exercises**). You should show this paper to your instructor when you are done.

As always, you should try to finish the lab during your section. However, if you do not finish during section, you have **until the beginning of lab next week to finish it**. You should always do your best to finish during lab hours; remember that labs are graded on effort, not correctness.

1. Understanding the Graphics Window

Before we get you started on this lab, we should first describe a bit how `DemoShapes` works. Notice that class `DemoShapes` extends `JFrame`, so that `DemoShapes` is associated with a window on the monitor. You saw a similar example to this in the lab last week.

It is possible to can draw pictures inside a `JFrame`. The JFrame window is a two-dimensional plane of pixels. The pixels are indexed by integer coordinates, as follows:

```
(0,0)  (1,0)  (2,0)  ...
(0,1)  (1,1)  (2,1)  ...
(0,2)  (1,2)  (2,2)  ...
...
```

In a pixel coordinate (x,y), x is the horizontal coordinate; it increases from left to right. Similarly, y is the vertical coordinate; it increases from top to bottom.

The method `paint(Graphics g)` is inherited from `JFrame`. Whenever the system knows that it needs to redraw the window, it calls this method `paint`, with a suitable `Graphics` object `g` as its argument. The object named in `g` contains several methods that can be used to draw, and these are the ones used in the program.

Method	Description
`g.drawLine(x1, y1, x2, y2);`	Draw a line from pixel `(x1,y1)` to `(x2,y2)`
`g.drawRect(x, y, w, h);`	Draw a rectangle with upper left corner `(x,y)`, width `w`, and height `h`
`g.getColor()`	Yields: the color currently being used to draw
`g.setColor(c);`	Set the color to draw with to `c` (which is of class `Color`)

---

2. Introduction to Abstract Classes

Problem 1: Preventing a Class from Being Instantiated. In some situations, we do not want programmers to instantiate (create an instance of) a particular class; rather we choose to define the class only so that it can serve as a default superclass of other classes. This is ideal for containers, when the container can only values that all have the same type.

For example, we might want to have a class Shape to serve as superclass for a number of "real" subclasses, like Parallelogram or Rhombus. We want to have Parallelogram and Rhombus objects around, but not generic "Shape" objects. However, so far we have not learned of a way to prevent programmers from instantiating (creating instances of) a class.

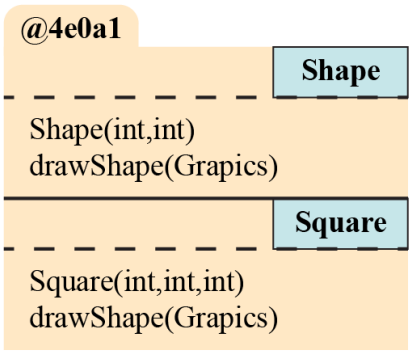
Solution: The correct approach is is to change the class to an **abstract class**; by definition and an abstract class cannot be instantiated. To do this, change the first line of the definition of the class, say class `C`, from

public class `C` { to **public abstract class** `C` {

Purpose of making a class abstract: You make a class abstract so that it cannot be instantiated (one cannot create an instance of it).

Problem 2: Forcing Programmers to Override a Method in a Subclass. In the abstract class `Shape`, the method `drawShape` is defined *only* so that it can be overridden by "real-shape" subclasses. We do not want programmers to call the `drawShape` method in `Shape`, and in fact, such a method would not make sense. How do we draw it if we do not know exactly what the shape is supposed to be?

Instead, the programmers should write overriding methods in the appropriate subclasses, which contain the drawing code for the specific shapes. However, we cannot force them to override the method in `Shape`, and if they do not, the `drawShape` method in `Shape` will be called. Therefore, we want something that will force programmers to implement a new `drawShape` method for each subclass of `Shape`.



Solution: The correct approach is to change `drawShape` to an abstract method, because abstract methods by definition *must* be overridden. To do this, change `drawShape` as shown below. There are two changes: (1) the keyword `abstract` is inserted and (2) the method body, including the enclosing braces, is replaced by a semicolon.

```
public void drawShape(...)  { ... }
                    to
public abstract void drawShape(...);
```

Purpose of making a method abstract: You make a method in an abstract class abstract so that it cannot be called and must be overridden.

---

3. Lab Exercises

Task 1: Make class `Shape` abstract. In file `DemoShapes.java`, place the following statement in method `paint`, just before the declaration (and initialization) of variable `h`:

```
Shape s0 = new Shape(5,5);
```

Compile the program. Write down what this statement does.

In the file `Shape.java`, place keyword `abstract` just before keyword `class` in the class definition, so that the third line of the file looks as follows:

```
public abstract class Shape {
```

You have made the class into an abstract class. Compile the program again. Do you get an error message? Write down the error message and explain in a few words why there is an error.

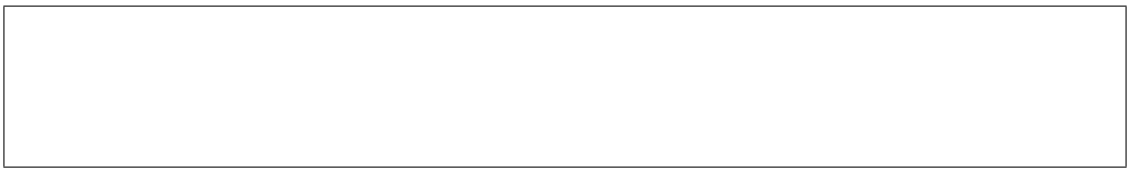
Now delete the statement that you placed in file `DemoShapes.java` above and compile again. You should no longer have an error message.

Task 2: Make method `drawShape` of class `Shape` abstract. In file Shape.java, change method `drawShape` to the following:

```
public abstract void drawShape(Graphics g);
```

Remember to replace the body {} by a semicolon. You have made this method into an abstract method. Compile the program; it should still compile.

Open file `Parallelogram.java` and comment out method `drawShape` (put `/*` before the method and `*/` after the method). Try to compile the program. Do you get error messages? Write on your paper the rror message that deals with class `Parallelogram`. Write a few words explaining what the error is.



Remove the comment symbols, so that `drawShapes` is again defined in `Parallelogram`. Try compiling the program again just to be sure that you removed them correctly.

Task 3: Add Some Arms. Class `Shape` is designed to be the root of all classes that draw a shape. We have the following hierarchy:

`Object` → `Shape` → `Parallelogram` → `Rhombus` → `Square`

because a square is a rhombus with 90-degree angles, a rhombus is a parallelogram all of whose sides are of equal length, and a parallelogram is a shape.

The shape that appears when a new DemoShapes is created looks almost like a person. It is drawn using methods of instance `g` of class `Graphics` that is attached to the `JFrame` object that opens. The only methods you need from `Graphics` are `setColor` and `getColor`. You will do most of your work in this task using the Shape classes.

You will give the person arms. All the changes you will make will be in class `DemoShapes`. Read through method `paint` of `DemoShapes`.

But before you do anything else, you should comment out the code that produces the two black lines (in DemoShapes).

Hint: look for where the color is set to black.

How to Draw the Arms. Each arm is should be a green rectangle that is 60 pixels long and 20 pixels high. Its leaning factor (field d of class Parallelogram) is 0, which means that it is a rectangle. The leaning factor is defined on Lesson page 4.4 of the ProgramLive CD (see also the comment at the beginning of class Parallelogram), but you really do not have to read about it. Later, when you get the program going with leaning factor 0, you can try a different leaning factor, say 15, and see what it looks like.

The arms should be attached at the top right and top left of the square that makes up the body. The tops of the arms should be parallel to the top line of the square.

In writing the code that draws these rectangles, use the variables that are defined at the top of method `paint`. Also, use variables to contain all the constants that you need, as we did in method `paint`. You may have to move the whole figure to the right (by changing the value of variable `x`) so that you can see the whole picture. You must use class `Parallelogram`; you may not use method `drawRect` in class `Graphics`.

Hint: to figure out the coordinates for the arms, look at the positioning of the green square. For debugging purposes, you may want to include System.out.println statements to see important values for the objects you create, like we did. The command System.out.println() outputs the given text (or toString() method if an object) to the console.

When you are finished, write on your paper the sequence of statements that you added to method `paint`.