CS1110 Lab 08. Abstract classes Fall 2011

Name ____________________ NetId ___________________________

This lab introduces you to the useful software-engineering concepts of "abstract class" and "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. Your lab instructor will present the concepts to you (which are quite simple) if you ask.

This lab also gives you practice in reading Java programs and, in the context of this lab, modifying them to draw some shapes in a JFrame. You will need a sheet of paper to write information about this lab on. Show it to your instructor when you have finished.

Problem 1. In some situations, we don't want programmers to instantiate (create an instance of) a particular class, but rather we choose to create the class only so that it can serve as a generic or default superclass of other classes. 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.

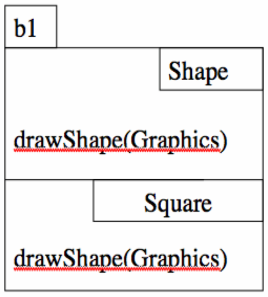
But so far we haven't learned of a way to prevent programmers from instantiating (creating instances of) a class.

Solution. Change the class to an **abstract class**, which by definition 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. Make a class abstract so that it cannot be instantiated (one cannot create an instance of it).

Problem 2. In abstract class `Shape`, to the right, method `drawShape` is defined ONLY so that it can be overridden by "real-shape" subclasses. We don't want programmers to call the `drawShape` method in `Shape`; they should write overriding methods in the appropriate subclasses. But we can't force them to override the method in `Shape`, and if they don't, the `drawShape` method in `Shape` will be called.

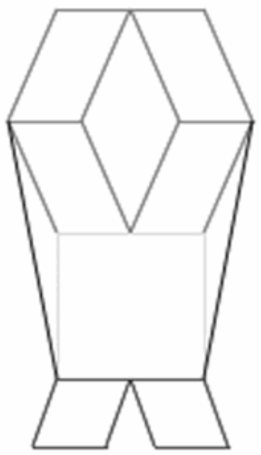


Solution. Change `drawShape` to an abstract method, because abstract methods by definition *must* be overridden. To do this, change `drawShape` as shown below. There are two changes: (1) keyword **`abstract`** is inserted and (2) the method body, including the enclosing braces, is replaced by a semicolon.

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

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

Step 1. Open some files in DrJava. Start a new directory on your hard drive. Download these five files into the directory: DemoShapes.java Shape.java Parallelogram.java Rhombus.java Square.java. You can also obtain them by opening the course web page in a browser and clicking "Labs" in the lefthand column; this opens a page that has links to these files.



Open files DemoShapes.java and Shape.java in DrJava and compile. In the Interactions pane, create an instance of DemoShape (this assumes you have unchecked the DrJava Interactions Pane preference "Require Variable Type"):

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

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

Short explanation of the graphics window used here

You can draw in a `JFrame`. Notice that class `DemoShapes` extends `JFrame`, so that `DemoShapes` is associated with a window on the monitor (you saw this on the second day of class). The JFrame window is a two-dimensional array with its pixels numbered as follows:

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

In a point (x,y), x is the horizontal coordinate. It increases from left to right. y is the vertical coordinate. It increases from top to bottom.

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. Object `g` contains several methods that can be used to draw, and these are the ones used in the program.

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

Step 2. Make class `Shape` abstract

(a) 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. On your paper, write what this statement does.

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

```
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` in part (a) and compile again. You should no longer have an error message.

Step 3. Make method `drawShape` of class `Shape` abstract

In file Shape.java, change method `drawShape` to:

```
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 error 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.

Step 4. Add 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 step 4 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`.

First, comment out the code that produces the two black lines (in DemoShapes). Hint: look for where the color is set to black.

Each arm is 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 don't 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.

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