This assignment introduces you to graphics. You will write procedures that draw a star, spirals, an expanding ball, and a "Koch snowflake" in a JFrame. You may work with one other person. If you do so, please form a group for this assignment on the CMS WELL BEFORE YOU SUBMIT YOUR FILES. You will not use a JUnit testing program because you will be looking at visual output (graphics) to determine correctness. You must use javadoc comments — before submitting the assignment, click the javadoc button and look carefully at the specs produced. At the end of this document, we tell you what to submit.

Download class Turtle (from here or from the course website), put it in its own directory, and open it in DrJava. A Turtle is a pen of a certain color at a pixel \((x, y)\) that is pointing in some direction, given by an angle (0 degrees is to the right, or east; 90 degrees, north; 180 degrees, west; and 270 degrees, south). When the turtle is moved to another spot using procedure move, a line is drawn if the pen is currently "down" --nothing is drawn if the pen is "up". The pen is initially black, but its color, of class java.awt.Color, can be changed. A footnote on page 1.5 of the ProgramLive CD contains information about class Color.

At this point, look at the specification of class Turtle (the javadoc files) and see what methods are available. Here are some important points:

- Class Turtle uses the Graphics object that is attached to a JPanel. It builds on class Graphics by maintaining the "turtle", which has a position and an angle. You can have many turtles open at the same time; they all use the same JPanel. This is because fields JFrame, panel, and graphics are static.
- The coordinates and angle of the turtle are maintained using type double. This is needed for maximum accuracy. If we used int, errors might crop up after many calculations. But whenever a point is to be placed in the window, its x- and y-coordinates are rounded to the nearest integer because that is what the graphics space needs.
- Function tColor allows you to use an integer to obtain an object of class Color, for various oft-used colors.
- Things are actually drawn on a "panel" on the JFrame. Dragging to make the JFrame window smaller or bigger does not change the size of this panel, which is \((\text{width}, \text{height})\), where width and height are two fields of class Turtle. Procedure setPanelSize can be used to make the panel as large as possible within the current window.
- Procedure moveTo \((x, y, a)\) can be used to move the turtle, without drawing, to \((x, y)\) and face it at angle \(a\).
- Procedure pause(p) can be used to pause execution for \(p\) milliseconds. Judicious use of this method will allow you to watch something being drawn in slow motion.

In DrJava, create another file with the following class in it (copy and paste; then use the indent-line feature of DrJava to indent the lines appropriately) and save it in the same directory with file Turtle.java.

```java
import java.awt.*;
/** Assignment A4: using a Turtle */
public class MyTurtle extends Turtle {
/** Draw a black line 30 pixels to the right (east) and then a red line 35 pixels down (south). */
public void drawTwo() {
    move(30); // draw a line 30 pixels to the right (east)
    addAngle(270); // add 270 degrees to the angle
    setColor(Color.red);
    move(35);
}
}
```

We give you one method in this class as an example of how graphics works. After compiling class MyTurtle, in DrJava's interaction pane, create an instance of class MyTurtle and then execute a call on this method and see what happens. A JFrame should be created and two lines should be drawn on it.

In the interactions pane (or in a method in class MyTurtle), draw some lines, rectangles, circles, etc, to familiarize yourself with class Turtle. After that, perform the tasks given below. Put a precise and complete specification on any method you write as a javadoc comment — many points will be deducted if you don't. The specification should allow anyone to know precisely what a call on the method does. It must mention all parameters and say what they are for. Look at your javadoc specs to make sure they are appropriate. As usual, your methods must have our names, exactly, and have the same parameters.
Task 1. Write a procedure `drawStar7(int d)` that draws a 7-pointed star whose lines have length \(d\) with the current turtle. An example appears to the right. Don't worry about what angle the turtle is facing when you start drawing the star; just begin by drawing a line of length \(d\). The angle between the two lines that make a point is \(180/7\) degrees. However, because of how the drawing is done, to get that angle in a point, after drawing a line, you have to add \(180 - 180/7\) degrees to the angle. For accuracy, the calculations MUST be done in type `double`.

Task 2: Draw a spiral. The first picture to the right is done by drawing 10 lines, as follows. The first one has length 5; the second, 10; the third, 15, etc. After each line, 90 degrees is added to the angle. The lines alternate among three colors: green, blue, red.

Write a procedure `spiral(int n, int a, int d, int msec);` that draws \(n\) lines, adding angle \(a\) after each one. Line 1 is \(d\) pixels long, line 2 is \(2*d\) pixels long, ..., line \(i\) is \(i*d\) pixels long. The lines alternate among green, blue, and red. Pause \(msec\) microseconds after drawing each line.

Then write a procedure `spiralm(int n, int a, int d, int sec)` that does the same thing as `spiral`, but first it:

1. Sets the size of the graphics panel as large as possible (use procedure `setPanelSize` in class `Turtle`).
2. Places the turtle in the middle of the window facing east.

When you first test your method, use 5 for \(d\) and 0 for \(sec\). Try different angles, like 90 degrees, 92 degrees, 88 degrees, etc. You can also use `msec` = 500 or `msec` = 1000 in order to see the lines drawn one at a time.

You will be amazed at what method `spiral` does. Find out by trying these calls (clear the panel, using procedure `clear()` , before each one — assume \(x\) is an instance of `MyTurtle`):

\[
\begin{align*}
\text{x.spiralm}(500, 90, 1, 0); & \quad \text{x.spiralm}(500, 135, 1, 0); & \quad \text{x.spiralm}(500, 60, 1, 0); \\
\text{x.spiralm}(500, 121, 1, 0); & \quad \text{x.spiralm}(500, 89, 1, 0); & \quad \text{x.spiralm}(500, 150, 1, 0); \\
\text{x.spiralm}(500, 120, 1, 0); & \quad \text{x.spiralm}(500, 119, 1, 0); \\
\end{align*}
\]

Task 3: An expanding-contracting ball. Procedure `fillCircle` in `Turtle` lets you draw a disk — i.e. a filled-in circle. You can use this method to draw what looks like a ball that expands and contracts at a rate given by a variable `velocity`. Basically, it will work as follows. You will write an infinite loop that at each iteration will do the following:

1. Pause for 100 microseconds.
2. If adding `velocity` to the radius will make the ball go into a side of the panel (touching the side is ok), then negate `velocity`.
3. If adding `velocity` to the radius will make the radius < 0, then negate `velocity`.
4. Add `velocity` to the radius and redraw the ball. To redraw the ball, if the velocity is negative, then the original ball must be erased (by drawing it white) before changing the radius and redrawing the ball.

That is how the expanding/contracting will work. Now, here is how to proceed to do this part.

(a) Start a new `.java` file for class `Ball`, which extends `MyTurtle`. This class needs two (private) fields: `radius` gives the radius of the ball and `velocity` the acceleration. You do not need getter methods for these fields.

(b) Write a constructor `Ball(r, v)` that initializes a new `Ball` folder so that the turtle starts in the middle of the window, facing east, and a ball of radius \(r\) is drawn in the center of the window. Later, when it is expanded, it will expand with velocity \(v\). Make sure this constructor works properly — by seeing that it constructs a ball properly in the window.

(c) Write a constructor `Ball (int r, int x, int y, int v, Color c)`. It should move the turtle to point \((x, y)\), facing east, change the pen color to \(c\), draw a ball with center \((x, y)\) and radius \(r\), and save \(v\) in field `velocity`.

(d) Write a procedure `expandOnce()` that adds `velocity` to the radius of the ball and redraws the ball. This method will assume (i.e. have as a precondition) that this step won't make a ball go over a side of the panel (because the ball gets too big) and it won't make the radius get < 0. Note that if `velocity` is negative, the procedure must do the following.

1. Erase the ball, by drawing it with a white pen; 2. add `velocity` to the radius; 3. draw the ball in its original color. So, you have to remember the original color. Have a local variable `save` to contain the original color and, after drawing the ball white, set the turtle color back to the value of variable `save`. 

Test this method carefully: Call it with both a negative and a positive velocity and make sure that the desired effect happens in the JFrame.

e) Write a procedure checkBall() that negates velocity (using velocity= - velocity;) if either of the following conditions holds. (1) velocity is negative, and adding it to the radius makes the radius be < 0. (2) velocity is positive, and adding it to the radius makes the ball go over a side (touching a side is OK).

e) Write a method expandContract() that perpetually expands and contracts the ball. Its body should be a loop that does not terminate and that has a repetend that (1) pauses for 100 microseconds, (2) reverses the expansion or contraction if necessary, as indicated in step (d), and (3) calls expandOnce. Test this procedure: In the interactions pane, create a new Ball d, call d.expandContract(); and watch the ball expand and contract. The best way to stop this execution is to hit the DrJava reset button.

Task 4. Koch snowflakes. Directly to the right is a triangle. It is called a "Koch snowflake " of level 0. If we replace each line of this Koch snowflake by four lines as drawn in the second diagram, we get the Koch snowflake of depth 1, which is the third figure. And, if we replace each of the lines of this Koch snowflake by the same four lines, we get the Koch snowflake of depth 2 shown on the extreme right. We can continue this to any depth.

Doing this is extremely easy using recursion! That's what you will do here. First, copy the two specs given below into your class MyTurtle.

/** Make sure that the graphics panel is as large as possible in the window. Clear the window, place the turtle facing east at position (width/5, 2*height/3). Then draw a Koch snowflake of depth d with line segment length F. Pause s milliseconds after drawing each line segment. Precondition: d ≥ 0 */
public void Koch(int d, double F, int s) {}

/** Draw a KochL snowflake of depth d with the current turtle. Parameters d, F, and s are as in procedure Koch.
Precondition: d ≥ 0 */
public void KochL(int d, double F, int s) {}

Procedure Koch is the easiest. First, call setPanelSize() in order make the graphics panel as large as possible within the window. Then, call procedure moveTo to move the turtle as in the specification (and make the turtle face east). Finally, draw as shown below.

The commands to be placed in Koch and KochL is described by the following two patterns:

- **Koch:** L -- L -- L
- **KochL:** L + L -- L + L

Here's how to interpret each of them. Each symbol is a command to do something, as follows.

L: If d (depth of recursion) is 0, draw a line of length F; otherwise, call KochL(d-1, F, s).
+: add 60 degrees to the turtle's angle.
-: subtract 60 degree from the turtle's angle.

So, the patterns are simply a simple, terse, way of describing what each of the method bodies should do. Such a system of patterns is called a "Lindenmayer System", after Aristid Lindenmayer, who co-authored a book titled *The Algorithmic Beauty of Plants* (Springer Verlag, 1990).

Your task, then, is to complete the bodies of Koch and KochL according to the patterns given above for them —and test and debug until they are correct. The following hint may help make things easier. In each procedure, first test whether d is 0; if it is, then carry out all the commands under that assumption and return. Then, carry out all the commands under the assumption that d > 0.
Extra Task! Hexagonal Gosper curves. Do this only if you want to! It is another example of writing recursive procedures using patterns. Doing it won't add to your grade, but we will make note of the fact that you did it. To the right are three "hexagonal Gosper curves" drawn using recursive procedures. The depth of recursion used to get the three was 0, 1, and 2. The first 2 were drawn using a line-segment length of 20; the third, line segment length 8. Three procedures are used to produce these curves as well as others with higher depths of recursion. Here are guidelines:

First, put into class MyTurtle three procedures with the following headers. You should be able to copy and paste — then, use the DrJava indent-line feature to make the definitions look readable.

/** Draw a Gosper design beginning at the position (4*width/5, height-50) of the window. d is the depth of recursion, F is the length of one line segment, and s is the microseconds to pause after each line segments id drawn. Precondition: d ≥ 0*/
public void Gosper(int d, int F, int s) {}

/** Draw a Left Gosper design. d, F, s are as in proc Gosper. Precondition: d ≥ 0*/
public void GosperL(int d , int F, int s) {}

/** Draw a Right Gosper design; d, F, s are as in proc Gosper. Precondition: d ≥ 0*/
public void GosperR(int d , int F, int s) {}

Procedure Gosper is the easiest. First, call setPanelSize() in order make the graphics panel as large as possible within the window. Then, call procedure moveTo to move the turtle as in the specification (and make the turtle face east). Finally, call GosperL with the same arguments as the parameters of Gosper.

GosperL and GosperR are mutually recursive procedures; they call each other. We can describe what they do by giving a pattern for each; here they are:

- GosperL: L + RR ++ R - L -- LL - R
- GosperR: - L + RR ++ R + L --L - R

Each symbol is a command to do something, as follows.

L: If d (depth of recursion) is 0, draw a line of length F; otherwise, call GosperL(d-1, F, s).
R: If d (depth of recursion) is 0, draw a line of length F; otherwise, call GosperR(d-1, F, s).
+: add 60 degrees to the turtle's angle.
-: subtract 60 degree from the turtle's angle.

Note that GosperL and GosperR call each other. We call them a set of two mutually recursive procedures. Now write their procedure bodies to follow the patterns given above. Test and debug until you believe that are correct.

What to submit. Before you submit, make sure classes MyTurtle and Ball are indented properly. Then, click the javadoc button and look at the API specs produced by that click. Check each method spec in MyTurtle and Ball to be sure that exactly what a call on the method does can be determined from the spec. If you don't do this, you will lose a lot of points.

Submit files MyTurtle.java and Ball.java.