FCPS Java Packets

Unit Two – Graphics and Animation

October 2018

Developed by Shane Torbert
edited by Marion Billington
under the direction of Gerry Berry
Thomas Jefferson High School for Science and Technology
Fairfax County Public Schools
Fairfax, Virginia

Contributing Authors
The author is grateful for additional contributions from Marion Billington, Charles Brewer, Margie Cross, Cathy Eagen, Philip Ero, Anne Little, John Mitchell, John Myers, Steve Rose, Ankur Shah, John Totten, and Greg W. Price.

The students' supporting web site can be found at http://academics.tjhsst.edu/compsci/
The teacher's (free) FCPS Computer Science CD is available from Stephen Rose (srose@fcps.edu)

Java Instruction Plan—Unit Two

Discussion

Panels and Frames

Robot worlds in JKarel were depicted in graphics windows. Graphics windows are created as instances of the class JFrame. A frame is a window with a border and a title bar. Content is made visible on frames using panels. Whatever you want the user to see is placed on a panel and that panel is placed on a frame.

The class hierarchies from the Java API are shown below. Fortunately, because of object-oriented programming, you don't have to know the details of each class! Each of your driver programs will create a frame. In your

```
java.lang.Object
  |
  +--java.awt.Component
        |
        +--java.awt.Container
              |
              +--java.awt.Window
                    |
                    +--java.awt.Frame
                          |
                          +--javax.swing.JFrame
```

```
java.lang.Object
  |
  +--java.awt.Component
        |
        +--java.awt.Container
              |
              +--javax.swing.JComponent
                    |
                    +--javax.swing.JPanel
                          ^
                          |
                       Panel00
```

graphics programs, each frame object will instantiate a panel object. The panel class that you make will inherit lots of functionality from the hierarchy. As you know, inheritance uses the keyword "extends":

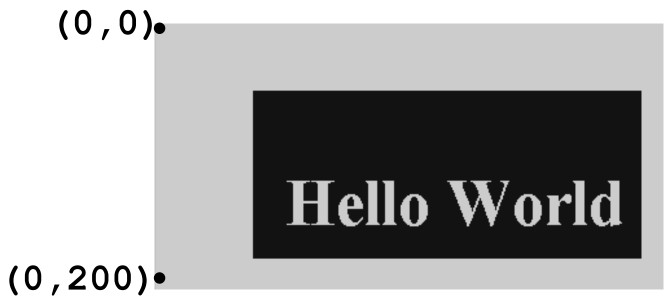
```
public class Panel00 extends JPanel
```

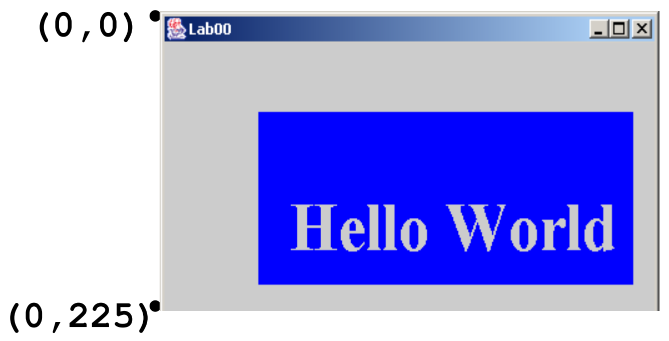
Notice that Panel00 isa JPanel isa JComponent isa Container isa Component isa Object.

Note that a frame is able to hold panels because each frame is also a container. Eventually we will use a panel's container ability to hold sub-panels.

```
new Panel00();
```

```
JFrame frame = new JFrame("Lab00");
frame.setContentPane(new Panel00);
```





The graphics coordinate system treats the upper-left corner of the panel as (0, 0). As you move horizontally to the right the x-values increase—this should be exactly as you are accustomed. As you move vertically *down* the y-values increase—this should be exactly *backwards* from what you are accustomed.

The reason the origin is placed at the upper-left corner rather than the lower-left corner is so that the origin remains fixed even when the user resizes the frame. Users typically manipulate the bottom-right corner in order to resize a frame.

Lab00
Hello World

Objective
The difference between a driver and a resource.

Background
In our graphics programs we have two files, a driver class and a resource class. The driver sets up the JFrame and instantiates the panel. The panel is the resource where all the action happens.

```
 4 import javax.swing.JFrame;
 5 public class Driver00
 6  {
 7   public static void main(String[] args)
 8    {
 9     JFrame frame = new JFrame("Lab00");
10     frame.setSize(400, 225);
11     frame.setLocation(100, 50);
12     frame.setDefaultCloseOperation(
                        JFrame.EXIT_ON_CLOSE);
13     frame.setContentPane(new Panel00());
14     frame.setVisible(true);
15    }
16  }
```

Line 4: The driver imports some classes from the `javax.swing` package.

Line 7: Each application begins at the `main` method.

Lines 9-14: Instantiates the frame object. Then it sends messages to that frame object.

Line 13: The frame gets its content from the panel object. The panel's `paintComponent` is called automatically.

```
 1  import javax.swing.*;
 2  import java.awt.*;
 3  public class Panel00 extends JPanel
 4  {
 5   public void paintComponent(Graphics g)
 6    {
 7     g.setColor(Color.BLUE);
 8     g.fillRect(75, 50, 300, 125);
 9     g.setFont(new Font("Serif",
                          Font.BOLD, 50));
10     g.setColor(new Color(150, 150, 0));
11     g.drawString("Hello World", 100, 150);
12    }
13  }
```

Lines 1-2: The panel class imports swing classes and Abstract Windowing Toolkit classes.

Line 3: Inherits lots of stuff from `JPanel`.

Line 5: Overrides `JPanel`'s `paintComponent` method. The `g` object has already been created somewhere up in the `JPanel` hierarchy. Since the code compiles, we know that `g` has access to all the methods of a Graphics object.

Lines 7-11: Sends messages to the `g` object. These messages are graphics commands. What does a graphics object know?

Graphics programs typically have lots of classes and objects. Make sure you save both your class files in the same folder, Unit2\Lab00. You may have to ask your teacher how to get the students' shells for Unit2.

Specification
Create both Unit2\Lab00\Driver00.java and Unit2\Lab00\Panel00.java. Enter the source code shown above, then compile and run the driver. After that works, experiment.

Sample Run
`Unit2Lab00.jar`

Lab00modify

Graffiti

Objective

To apply your understanding of color, font, and `drawString` commands.

Color

In Lab00 you created a new Color object (it was gold) with the constructor `new Color(150, 150, 0)`. Indeed, you can create any color by specifying the red-green-blue values, which are integers between 0 and 255. There are many color-pickers online. Here is one: http://www.colorspire.com/rgb-color-wheel/ Notice that any color is a combination of red-green-blue values. (0, 0, 0) is black. (255, 255, 255) is white. In hexadecimal values, black is #000000 and white is #FFFFFF. Play with the color-picker.

Java's `Color` class also predefines thirteen color constants: BLACK, BLUE, CYAN, DARK_GRAY, GRAY, GREEN, LIGHT_GRAY, MAGENTA, ORANGE, PINK, RED, WHITE, and YELLOW. To set the color to red, you may use either `g.setColor(Color.RED);` or `g.setColor(new Color(255,0,0));`

To set a darker shade of red: `g.setColor(Color.RED.darker());`

To set a brighter shade of red: `g.setColor(Color.RED.brighter());`

After you set a color, everything is drawn or written in that color until you change it.

Font

The Font constructor takes three arguments, the style, whether plain, bold, or italic, and the size. To set a Serif, plain, small font, use `g.setFont(new Font("Serif", Font.PLAIN, 8));`

To create a font object that is SansSerif, bold, and large, use:

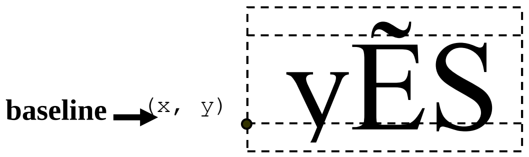
```
     Font f2 = new Font("SansSerif", Font.BOLD, 20);
      g.setFont(f2);
```

Two more examples:

```
     Font f3 = new Font("Monospaced", Font.ITALIC, 12);
     Font f4 = new Font("Arial", Font.BOLD | Font.ITALIC, 60);
```

Words or Text

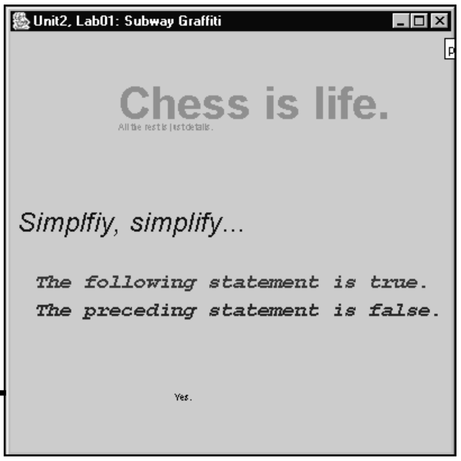
`drawString` takes three arguments, a String and an (x, y) pair of coordinates, which specify the position of the baseline of the string, as shown.



Specification

Create Unit2\Lab00\Driver00.java. Modify the file as needed.

Create Unit2\Lab00\Panel00modify.java. You may save Panel00.java as Panel00modify.java and modify the file. Display your favorite phrases, sayings, or quotes in the graphics window. Use at least four different colors, at least one user-defined color, at least four different fonts, all the possible font styles, and at least four different sizes.


Unit2, Lab01: Subway Graffiti

Chess is life.
All the rest is just details.

Simplfiy, simplify...

The following statement is true.
The preceding statement is false.

Yes.

Exercises
Lab00

Answer these questions about the Java API. The API (Application Programming Interface) is a document which lists and describes all you need to know about certain classes. The Java API for your version of Java is available on-line through any web browser. In jGrasp, go to Help|Java API.

1) What is a "constructor"?

2) How many constructors does JPanel define? ____

3) What does it mean "to implement an interface"?

4) How many interfaces does JPanel implement? ____

5) What are "instance methods"?

6) How many instance methods does JPanel define? ____

7) What is a "superclass"?

8) What is the superclass of Graphics? _____________

9) What is a "subclass"?

10) How many known subclasses does Graphics have? ____

11) Is *g* actually a Graphics object? Y/N How do you know?

12) What is an "argument" (in Computer Science)?

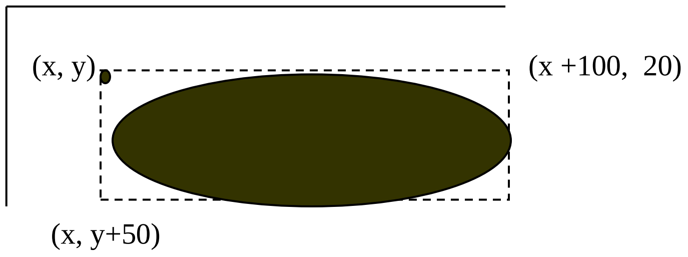
13) How many arguments does the `drawLine` method require?____

14) Why should you not use the method `getClipRect`?

15) Define "deprecated."

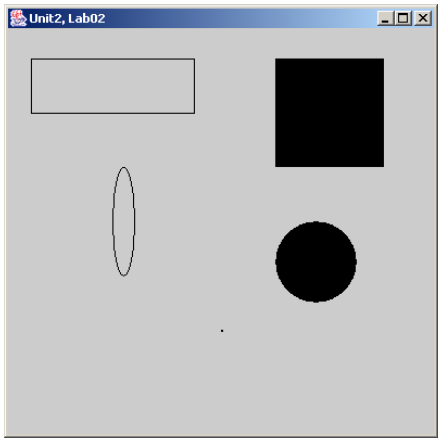
Discussion
Drawing Shapes

Many of the drawing methods take four arguments, (x, y, width, length). The (x, y) establishes the absolute upper-left corner of the shape, and (width, length) are relative from that. The diagram shows example `fillOval(20,20,100,50)`



The following shapes take four arguments. Note the filled form of the commands to produce filled rectangles and ovals. Make sure you know how the numbers work!

Rectangle	`g.drawRect(50, 50, 400, 75);`
Square	`g.fillRect(500, 50, 200, 200);`
Ellipse	`g.drawOval(200, 200, 20, 80);`
Circle	`g.fillOval(500, 300, 100, 100);`
Point	`g.drawRect(400, 600, 1, 1);`

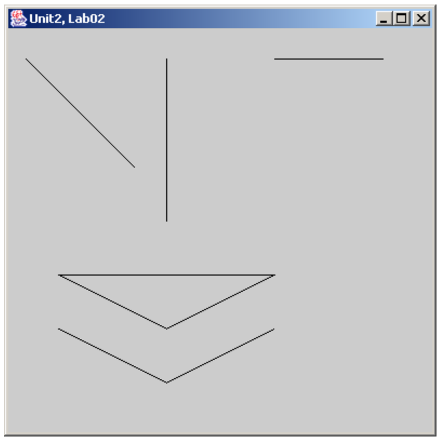


In these shapes, the pairs of arguments refer to (x, y) coordinates that are endpoints of the line segments. All four coordinates are absolute.

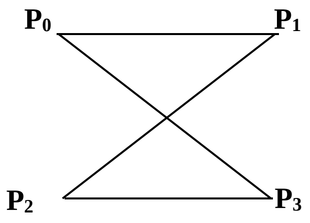
Line	`g.drawLine(20, 50, 120, 150);`
Vertical Line	`g.drawLine(150, 50, 150, 200);`
Horizontal Line	`g.drawLine(250, 50, 350, 50);`

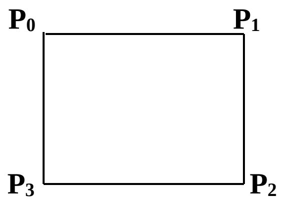
Polygon	`int xPoints[] = {50, 150, 250};`
	`int yPoints[] = {250, 300, 250};`
	`g.drawPolygon(xPoints, yPoints, 3);`

Polyline	`int xxPoints[] = {50, 150, 250};`
	`int yyPoints[] = {300, 350, 300};`
	`g.drawPolyline(xxPoints, yyPoints,3);`



Note that the third argument to polygon and polyline is the number of points that make up the polygon or polyline. This value will be equal to the length of each array `xPoints` and `yPoints`. The difference in these two commands is that a polygon automatically connects the last point back to the first; a polyline does not. Be careful! The *order* of the points for a polygon makes a difference. For instance:





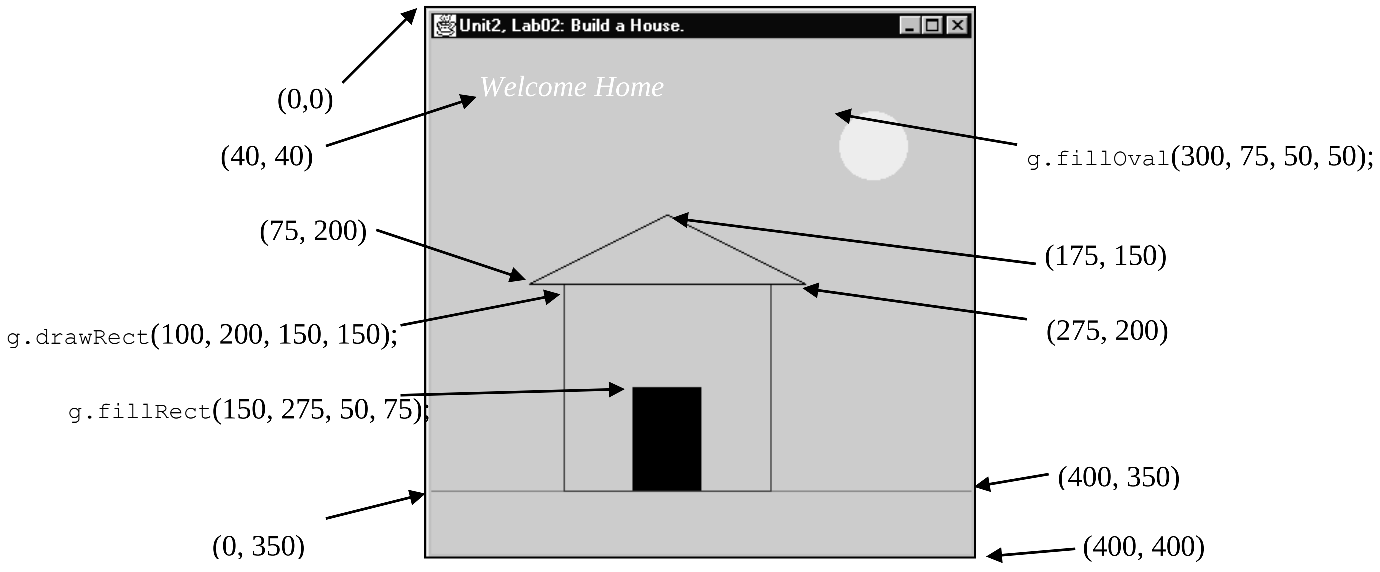
Lab01
Welcome Home

Objective

To create graphics using Java's drawing commands.

Specification

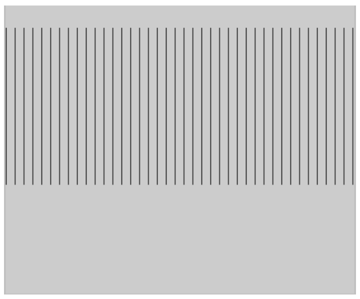
Create a new Java driver and panel in folder Unit2\Lab01. Reproduce the house below. Use the given coordinates. To see the colors, please run the .jar file.


Unit2, Lab02: Build a House.
Welcome Home
(0,0)
(40, 40)
g.fillOval(300, 75, 50, 50);
(75, 200)
(175, 150)
g.drawRect(100, 200, 150, 150);
(275, 200)
g.fillRect(150, 275, 50, 75);
(400, 350)
(0, 350)
(400, 400)

Sample run

`Unit2Lab01.jar`

Discussion
Loops and graphics



Loops can be used to draw repeating shapes. The code below draws the repeating vertical lines as shown at the right.

```
for(int x = 0; x <= 400; x += 10)
  g.drawLine(x, 25, x, 200);
```

By looking at the code, how do you know they are vertical lines?

Note that the *y*-values of the endpoints are always 25 and 200. The *x*-values of the endpoints are always the same number (for each line), and this number increases by 10 with each iteration of the loop, producing *x* values of 0, 10, 20, 30, …, 400. These two lines of code draw the forty-one lines you see on the panel. It is as if we had written out forty-one different `drawLine` commands, but it only took us two lines of code. Oh, the power of a for-loop!

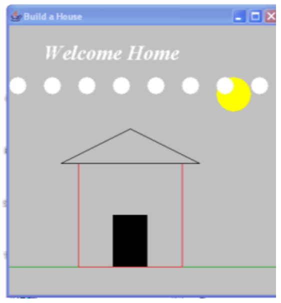
Lab01 continued
Welcome Home with loops

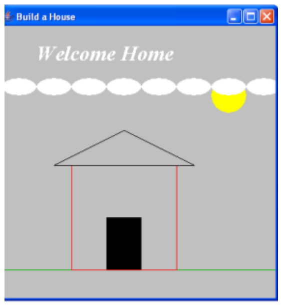
Specification

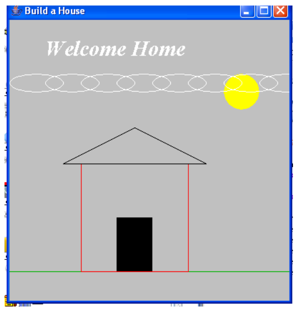
1. Return to Lab01 and draw a picket fence, using a for-loop, in front of your house in Lab01. When you finish, write the for-loop for your picket fence here:

2. Use a for-loop to draw repeated horizontal ovals for clouds. When you finish, write the code for your clouds:

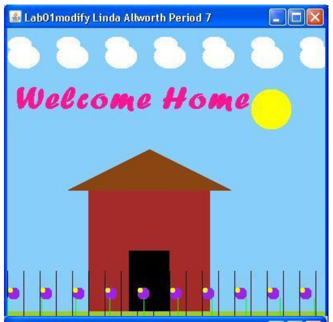
3. Write the for-loops that produced the horizontal clouds shown below.

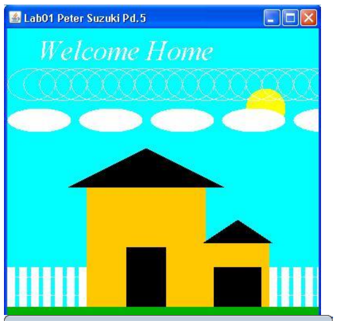


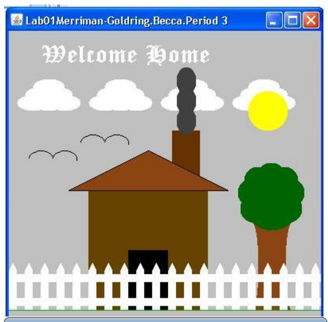


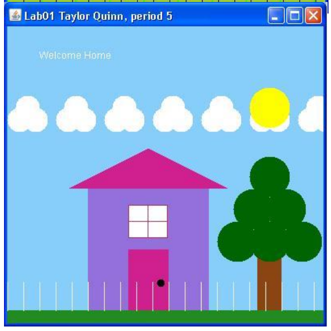


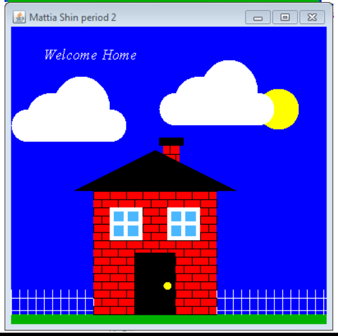
4. Now return to the house in Lab01 and make it fancy. Use your imagination. Here are some pictures that students have produced in the past. These pictures, in color, are also in your Unit 2, Lab01 folder.

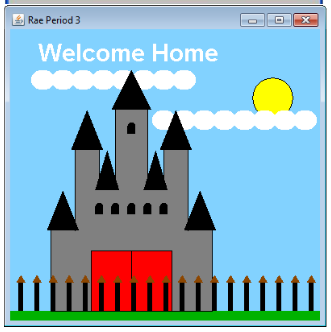












Exercises
Lab01

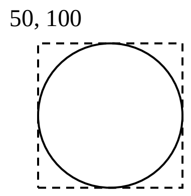
Using the Graphics object `g`, create the following shapes:

1) Draw a rectangle whose top-left corner is at position 50, 75 and whose width is 100 and height is 200.

2) Draw a square whose top-left corner is at position 200, 150 and whose sides have a length of 100.

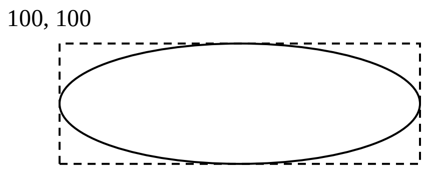
3) Draw a line connecting the points 20, 40 with 120, 80.

4) Draw a circle at position 50, 100 (position of top-left corner of imaginary rectangle in which the circle will be placed) with a width and height of 60.



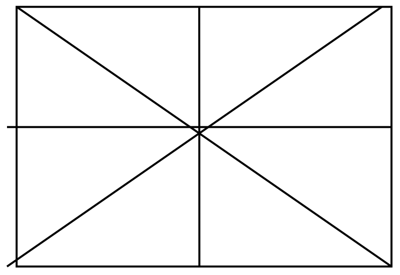
5) Now draw that same circle except use 50, 100 as its *center* point.

6) Draw an ellipse at position 100, 100 (position of top-left corner of imaginary rectangle in which the ellipse will be placed) with a width of 80 and height 30.



7) Now draw that same ellipse except use 100, 100 as its *center* point.

8) Using your knowledge of the `drawRect` and `drawLine` methods, create the following figure whose top-left corner is at 20, 20 and whose width is 100 and height is 80. (Hint: First write the commands to draw the rectangle, then write the commands to draw the four lines inside the rectangle.) It might help to label the beginning and ending points of each of the 4 cross lines.



Lab02
Our Fearless Leader

Objective
`drawImage` and `fillOval`

Background
Java uses an `ImageIcon` object to store a `jpg`, `jpeg`, `gif`, or `png` file:

```
   ImageIcon thomas = new ImageIcon("tj.jpg");
```

For this to work there must be an image file named `"tj.jpg"` in your Unit2\Lab02 folder. This image can be displayed with its upper-left corner at (50, 50) by using one of the commands:

```
   g.drawImage(thomas.getImage(), 50, 50, null);          //original size
   g.drawImage(thomas.getImage(), 50, 50, 25, 75, null);  //scaled image
```

The arguments `25` and `75` scale the image to be 25 pixels wide by 75 pixels high. The `null` is a place-holder for an object that we don't care to instantiate. In this case, `null` means about the same as "empty."

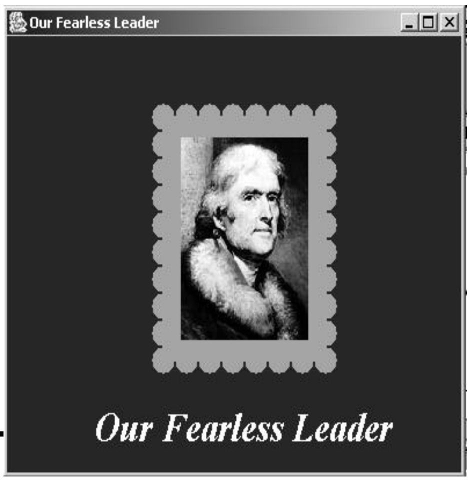
Note how different types of objects work together in order to make the image appear on the screen:

ImageIcon		Image		Graphics		JPanel		JFrame
object `thomas` is created	→	`thomas` knows its own image	→	`g.drawImage` gets thomas's image	→	`g` paints the panel	→	the panel appears on the screen

The `fillOval` method can be used to draw circles. The method requires four arguments. The first two arguments specify the upper-left corner of the rectangle (or square) enclosing the oval (or circle). They do not specify the center of the circle. The last two arguments specify the width and height of the enclosing square, also known as the circle's diameter. The last two arguments do not specify the radius of the circle. On the other hand, we often want to draw circles from the center with a certain radius. For a general circle with center at (x, y) and radius r, you may use: `g.fillOval(x - r, y - r, 2 * r, 2 * r);.` Of course, to use this formula, your code must assign concrete values to x, r, and y.

Be careful! The compiler will not compile 2r. You must write 2 * r.

If you can make the x change regularly, using a for-loop, you can draw a horizontal row of repeated circles for a frame effect. How do you draw a vertical row of circles?



Specification
Run `Unit2Lab02.jar`. Duplicate the image. The frame is gold and the background is red.

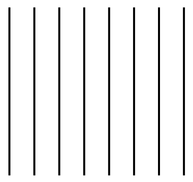
Create filename Unit2\Lab02\Driver02.java. Make sure to add a panel object of type Panel02.

Create filename Unit2\Lab02\Panel02.java. Use graphics commands to draw a framed picture of your school's namesake, principal, or inspirational leader. Estimate all distances by trial-and-error.

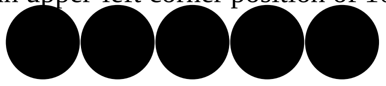
Exercises
Lab02

Using the Graphics object `g`, create the following shapes:

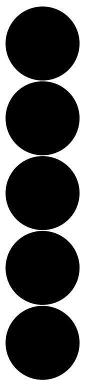
1) Use a `for`-loop to create the following image. Each line is 50 pixels long and 5 pixels apart. The top of the first line is at 80, 50.

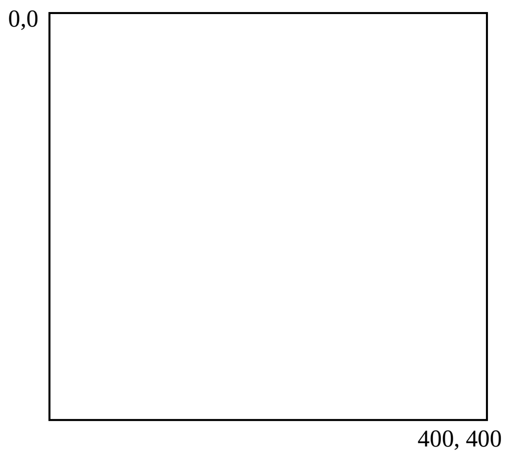


2) Use a `for`-loop to create the following image. Each circle is green, 20 pixels wide, and the first circle on the left has an upper-left corner position of 100, 0.



3) Use a `for`-loop to create the following image. Each circle is magenta, 20 pixels wide, and the top circle has an upper-left corner position of 100, 100.





4) Draw the output of this command:

```
for(int y = 0; y < 5; y++)
    g.drawLine(200, 0, y*50, 200);
```

5) Draw the output of this command:

```
for(int y = 20; y <= 80; y+=20)
  g.fillOval(20, y, 20, 20);
```

6) Draw the output of these commands:

```
int y = 0;
for(int x = 0; x < 400; x+=50)
  g.drawOval(x, y+=50, 50, 50);
```

Discussion
Buffering an image

Up to now, we drew graphics directly on the panel by sending commands to the `g` object in `paintComponent`. Starting with Lab03 we will draw the image in an off-screen *buffer*, or temporary storage area, and then "paste" the complete image on the screen once at the end. Drawing to a buffer is much faster than drawing to the screen. When we start animation, with lots of drawing commands, using the buffer eliminates images that flicker.

Lines 9 and 10 declare two *private fields* (variables) in this object, `N` and `myImage`. Line 9 also declares `N`, which is the size of the panel, as `final`, meaning that `N` is a *constant* in the code. (`N` is a constant, even though a user can change the size of panel by clicking and dragging a corner.) Line 10 creates a reference but does not assign an object. Line 12 begins the constructor. Line 14 actually instantiates and assigns the `BufferedImage` object to the reference that is on Line 10. Line 15 gets the Graphics object that is inside the `BufferedImage` object and assigns a reference called `buffer`. Beginning on Line 16, we send the familiar drawing, color, font, and `drawString` commands to the `buffer` object.

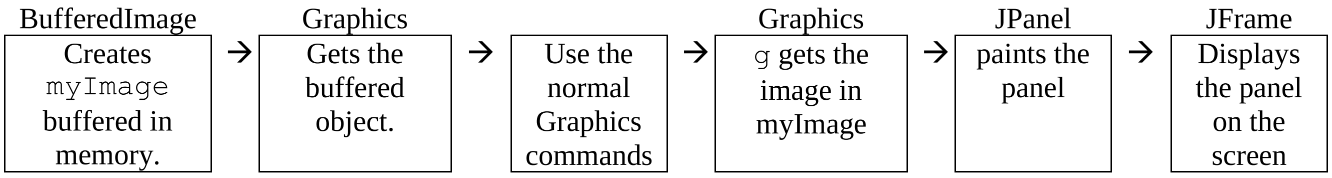
```
 6  import java.awt.image.BufferedImage;
 7  public class Panel03 extends JPanel
 8   {
 9    private final int N = 400;
10    private BufferedImage myImage;
12    public Panel03()
13     {
14        myImage = new BufferedImage(N, N, BufferedImage.TYPE_INT_RGB);
15        Graphics buffer = myImage.getGraphics();
16        buffer.setColor(Color.BLUE);
17        buffer.fillRect(0, 0, N, N);
```

All the graphics commands work exactly as you would expect them to. The only difference is that you are sending messages to the buffered object. In order to actually see what you've drawn, you must "paste" the buffered image onto the panel, using one command in `paintComponent`:

```
public void paintComponent(Graphics g)
{
   g.drawImage(myImage, 0, 0, getWidth(), getHeight(), null);
}
```

The methods `getWidth` and `getHeight` return the current width and height of the panel. These values may change during runtime if the user resizes the frame. When you resize, the image will change to fill up the panel.

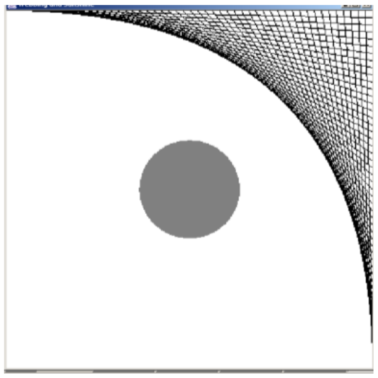
The different objects work together in this fashion:


BufferedImage
Creates myImage buffered in memory.
Graphics
Gets the buffered object.
Use the normal Graphics commands
Graphics
g gets the image in myImage
JPanel
paints the panel
JFrame
Displays the panel on the screen

There is a shell for Lab03 that has this off-screen buffer already set up for you. Don't change it!

Lab03
Webbing and Sunshine

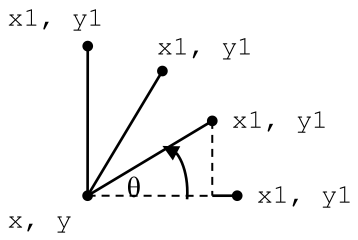
Objective
Calculating end-points and using `drawLine`



Background
The first task is to draw webs at the corners. This for-loop draws a web in the upper-right corner of the N x N image:

```
   for(int k = 0; k <= 50; k++)
   {
      buffer.drawLine(N * k / 50, 0, N, N * k / 50);
   }
```

Notice that 51 lines will be drawn. `drawLine` takes two pairs of coordinates, the end points of each line. One endpoint of each line needs to move from left to right across the top of the image while the other endpoint needs to move from top to bottom down the right side of the image. If we can make x increase, then (x, 0) will slide the top endpoint over. At the same time, an increasing y will slide (N, y) down. We can make x and y increase in a for-loop, for as k increases from 0 to 50, the expression N * k / 50 increases from 0 to N in 50 equal increments. Thus, the calculated coordinates (N * k / 50, 0) and (N, N * k / 50) together slide the x-coordinate of the first point along the top and the y-coordinate of the second point down the right side. (Warning! Due to something called "integer division," the expression k / 50 * N will evaluate to 0. Make sure to multiply N * k before dividing by 50.) You'll have to figure out how to move the endpoints for the other three corners.



The second task is to draw the rays of the sun. The center is at (x, y) but each of the other endpoints must be calculated. From geometry, we know that in any right triangle the sine of an angle is the ratio of its opposite side to its hypotenuse and the cosine of an angle is the ratio of its adjacent side to its hypotenuse. This means that if we know the angle and the length of the hypotenuse, we can determine the lengths of the opposite and adjacent sides, which are the coordinates (x1, y1). The angle θ changes from 0 to 360 in equal increments, depending on how many rays we want. We'll use a for-loop to change the angle, of course. Inside the for-loop we will calculate the (x1, y1) coordinates and draw the line.

It happens that Java's `Math.cos` and `Math.sin` methods require radian arguments. Since most people prefer to think in degrees, we multiply degree measures by Math.PI / 180. Also, these trigonometry methods return decimal values, so we need to *cast* the coordinates to integers using `(int)`. For example:

```
   x1 = (int)(x + size * Math.cos(angle * Math.PI / 180));
```

Specification
Create Unit2\Lab03\Driver03.java. Make an appropriate driver, then compile and run.

Load Unit2\Lab03\Panel03.java. Complete the definition of the constructor to draw the full, four-cornered webbing and a sun with rays. Draw 12 rays at first, but make it easy to draw any number of rays.

Extension
Change all constants into variables. Prompt the user to enter N, the number of lines, and the number of rays. You will need `Integer.parseInt`, which turns a string into an int.

Sample Run `Unit2Lab03.jar`

Exercises
Lab03

Complete these questions using the Java API.

1) A *field* in Java is a variable created and used by a class. How many fields are in the Math class?____

2) What two numbers are stored in the fields of the Math class?

3) The Math class fields are marked **`public static final double`**. Explain what each word means.

`public`

`static`

`final`

`double`

4) Explain why it makes sense that π and *e* are **`public static final double`** in Java.

5) How many constructors does the Math class define? ______Why does that make sense?

6) What does a **`void`** method do?

7) How many void methods does the Math class define? ____

8) Explain the Computer Science meaning of "to pass an argument."

9) How many no-argument methods does the Math class define? ________ Name them ________________

10) Why can't you create a Number object?

11) How many direct known subclasses does Number have? _______

12) How many interfaces does Integer implement? _______ Name them _______________________________

13) How many constructors does Integer define? ________

14) What do you think an Integer constructor does?

Lab04
Buckets

Objective
Class methods vs. instance methods

Background
Here is a classic logic problem: given two buckets, a 3-gallon bucket and a 5-gallon bucket, how can you measure out exactly 4 gallons?

```
 5 import javax.swing.*;
 6 import edu.fcps.Bucket;
 7 public class Driver04
 8 {
 9  public static void main(String args[])
10  {
11   JFrame frame = new JFrame("Buckets");
12   frame.setSize(600, 400);
13   frame.setLocation(100, 100);
14   frame.setDefaultCloseOperation(
                    JFrame.EXIT_ON_CLOSE);
15   frame.setContentPane(new BucketPanel());
16   frame.setVisible(true);
17   Bucket.setSpeed(1);
18   Bucket.useTotal(false);
19   Bucket five = new Bucket(5);
20   Bucket three = new Bucket(3);
21   //  implement the algorithm here
22
```

The algorithm to implement is:

1. Fill the five-gallon bucket.
2. Pour from five into three until three is full.
3. Empty the three-gallon bucket.
4. Pour from five into three until five is empty.
5. Fill the five-gallon bucket.
6. Pour from five into three until three is full.
7. Done! The amount left in five is exactly 4.

The `edu.fcps.Bucket` class gives you the tools to program a solution. As you might expect, the Bucket class allows you to instantiate bucket objects, to fill them up, and to pour them out. The class also draws the buckets on a table, shows the water levels rising and falling, and tracks the number of gallons by turning white numbers to red. You look up the methods in the Bucket class by going to its API at https://academics.tjhsst.edu/compsci/CSweb/index.html .

On line 17, we see the *class method* `Bucket.setSpeed(1)`. We know it is a class method because the dot notation is invoked on the class Bucket. The compiler knows it is a class method because of the keyword `static`

```
public static void setSpeed(int s)
public static void useTotal(boolean b)
```

```
Bucket class
   setSpeed(int s)
   useTotal(boolean b)
```

It should make sense that `setSpeed` would be owned by the Bucket class, not by the individual bucket objects. It should also make sense that the method `pourInto` would be owned by the individual bucket objects. If the individual objects own the method, it is called an *instance method*. The compiler knows it is an instance method because there is no `static`, e.g.

```
public void pourInto(Bucket arg)
public void fill()
public void spill()
```

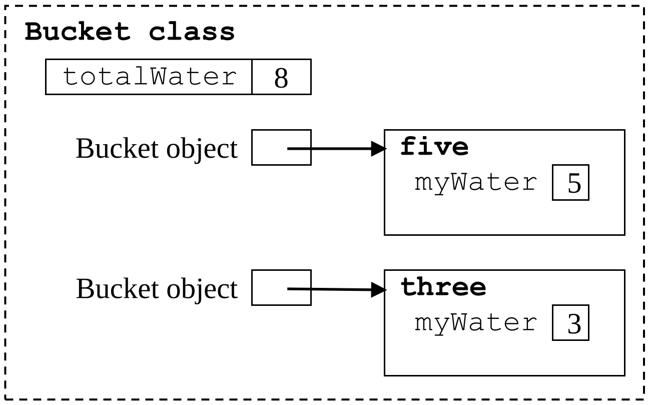
```
a Bucket object
   pourInto(Bucket b)
   fill()
   spill()
```

Static methods add powers to the class, not to the objects. Static methods remain with the class and do not get duplicated in each object. In consequence, there will be exactly one copy of `setSpeed` and it is owned by the Bucket class. However, there may be many of copies of `pourInto`, because each Bucket object owns its own copy of `pourInto`.

Specification: Write Driver04 to implement the algorithm to get exactly 4 gallons. Don't change BucketPanel!
Sample Run `Unit2Lab04.jar`

Lab04
Buckets, Part II

Objective
Class fields vs. instance fields



Background
We instantiate two Bucket objects with:

```
      Bucket five = new Bucket(5);
      Bucket three = new Bucket(3);
```

The first part of the code in the Bucket class looks like:

```
public class Bucket
{
  private static int totalWater;
  private int myWater;
  ...
}
```

The keyword `static` causes the computer to create one copy of the static field `totalWater` and give it to the Bucket class. The lack of `static` causes the computer to create two copies of `myWater` and give one to each Bucket object. The Bucket class stores the total water (8) in the system in `totalWater`, while each Bucket object stores its own value in `myWater` (5 and 3).

On Line 18 in Driver04, you called the class method `Bucket.useTotal(false)`. The effect of this mystery command was to turn off the static field `totalWater`. In consequence, the program kept track of the numbers in the instance fields of each bucket object. When either bucket held a certain number of gallons, it turned that number from white to red. Run it again and notice when the little white numbers turn red.

In contrast, line 18 in Driver04a has a call in to `Bucket.useTotal(true)`. The effect of this mystery command is to turn on the static field `totalWater`, which keeps track of the total volume of water in all the buckets. In this case, when the water in the system holds a certain number of gallons, that number turns from white to red.

```
 5 import edu.fcps.Bucket;
 6 import javax.swing.*;
 7 public class Driver04a
 8 {
 9  public static void main(String args[])
10  {
11   JFrame frame = new JFrame("Buckets");
12   frame.setSize(600, 400);
13   frame.setLocation(100, 100);
14   frame.setDefaultCloseOperation(
                        JFrame.EXIT_ON_CLOSE);
15   frame.setContentPane(new BucketPanel());
16   frame.setVisible(true);
17   Bucket.setSpeed(5);
18   Bucket.useTotal(true);
19   Bucket five = new Bucket(5);
20   Bucket three = new Bucket(3);
21   // implement the algorithm here
22
23
24
```

Specification
Modify `Driver04`, calling it `Driver04a`. This time track the total volume of water. Turn the white numbers to red, showing that the system can measure out 1, 2, 3, 4, 5, 6, 7, and 8 gallons.

Sample Run `Unit2Lab04a.jar`

Extension Create a new driver named `Driver04b`. Make three Bucket objects, of 3, 4, and 5 gallons. Track the total volume of water by turning the white numbers to red. Show that the system can measure out 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, and 12 gallons.

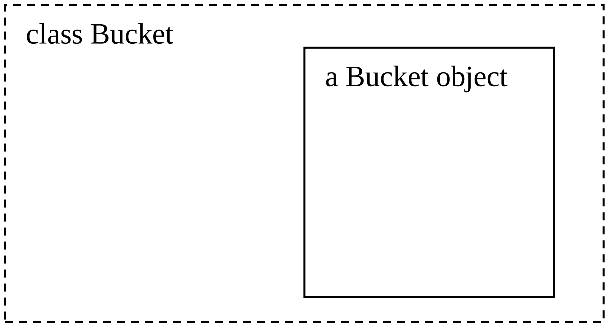
Exercises
Lab04

1) Given two buckets named five and six with capacities of five and six, respectively, produce exactly four gallons of water.

2) Given two buckets named four and five with capacities of four and five, respectively, produce exactly two gallons of water.

3) Here is the beginning of class `Bucket`:

```
class Bucket
{
  private static int totalWater = 0;
  private static int numBuckets = 0;
  private final int myCapacity;
  private int myWater;
  . . .
}
```



3a) Draw and label the class fields and the instance fields. Use the boxes above.

3b) What's the purpose of `totalWater`? ______________________Who owns `totalWater`? ____________

3c) What's the purpose of `myCapacity`? ______________________ Who owns `myCapacity`? ___________

3d) What's the purpose of `myWater` ? ____________________________

3e) The method `fill` has to change two fields. What are they? _________________and ________________.

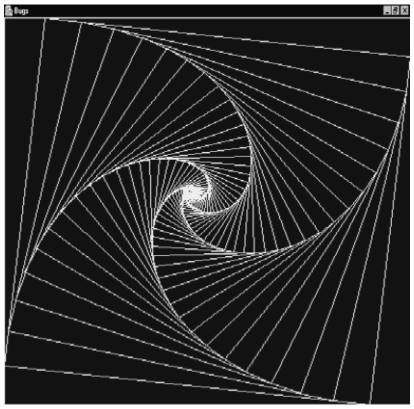
3f) Each Bucket object has two private fields, `myWater` and `myCapacity`. When does the value of `myWater` change?

3g) Each Bucket object has two private fields, `myWater` and `myCapacity`. When does the value of `myCapacity` change?
When does the value of `myCapacity` get set?

4) Explain why it makes sense that `setSpeed` is a static method.

5) A certain class has the code for methods for **`public`** `void aaa()` and **`public static`** `void bbb()`. In the driver, you instantiate three objects from this class. How many copies of method `aaa` do you have? ____
How many copies of method `bbb` do you have? ____

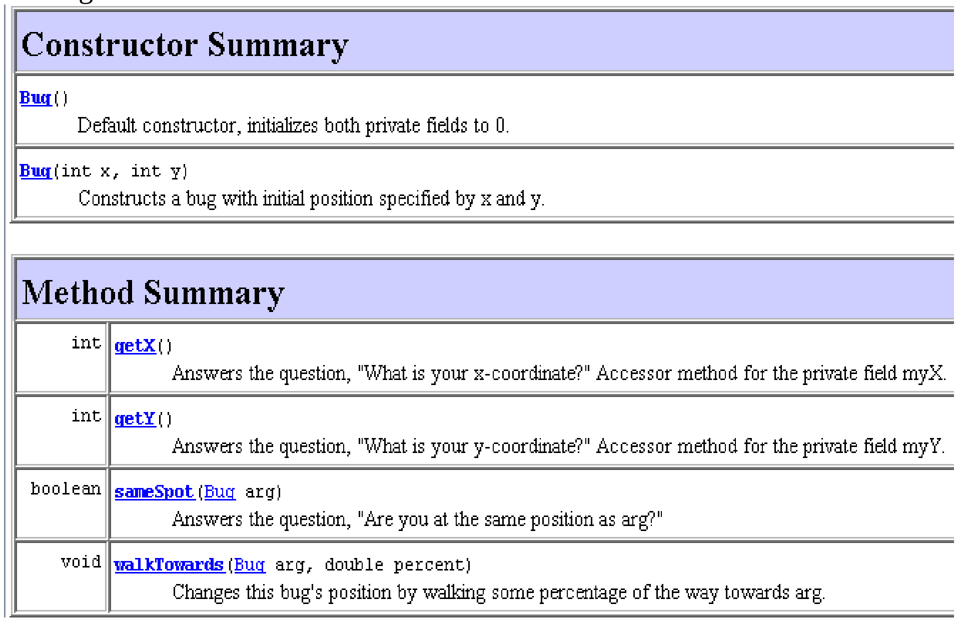
Lab05
Bugs



Objective
An example resource class, Bug.

Background
The Bug class provides most of the tools to draw the spiral design. Like most of our classes, a Bug has *private* data and *public* methods. The private data is hidden from the user, but the public methods make that data available to the user. Programmers do it this way because it makes the code safer and easier both to use and to write.

The API lists the public methods, which is what the user needs to know in order to use the object. How do you instantiate bugs? By calling one of its constructors—of course!



Constructor Summary

`Bug()` Default constructor, initializes both private fields to 0.
`Bug(int x, int y)` Constructs a bug with initial position specified by x and y.

Method Summary

`int`	`getX()` Answers the question, "What is your x-coordinate?" Accessor method for the private field myX.
`int`	`getY()` Answers the question, "What is your y-coordinate?" Accessor method for the private field myY.
`boolean`	`sameSpot(Bug arg)` Answers the question, "Are you at the same position as arg?"
`void`	`walkTowards(Bug arg, double percent)` Changes this bug's position by walking some percentage of the way towards arg.

What can bugs do? A Bug can tell you where it is. Each Bug has two public *accessor methods* `getX` and `getY`. Each returns an `int`.

What else can bugs do? A bug can report, either `true` or `false` (which are *boolean* values), whether it is at the same spot as another bug. You don't know how the *instance method* `sameSpot` works, but you do need to know that it takes one argument, a Bug, and returns a boolean.

What else can bugs do? A bug can walk a percentage of the distance towards another bug. The *instance method* `walkTowards` requires two arguments, the other bug (the `Bug arg`) and a decimal number (the `double percent`) of the percentage of the distance to travel. It is a void method because it does not return anything.

Are bugs able to draw lines? Look at the API. No, bugs do not know how to draw lines. You have to invoke that command yourself on the buffer.

Specification
Load Filename Unit2\Lab05\Driver05.java and \Bug.java. Don't change these!
Load Filename Unit2\Lab05\BugPanel.java. Implement the algorithm below:

Steps in the algorithm:	Write an example that calls the appropriate method:
1. Instantiate 4 bugs, one at each corner.	
2. Tell each bug to move 10% of the way toward its clockwise neighbor.	
3. Draw a line between each bug.	
4. Repeat steps 2 and 3 until any two bugs are at the same spot.	

Sample Run `Unit2Lab05.jar`

Exercises
Lab05

On the previous page we studied the API for Bug. Now study the *source code* for Bug. What do bugs know? What can they do to themselves?

```
public class Bug
{
  private int myX;
  private int myY;
  public Bug()
  {
    myX = 0;
    myY = 0;
  }
  public Bug(int x, int y)
  {
    myX = x;
    myY = y;
  }
  public int getX()
  {
    return myX;
  }
  public int getY()
  {
    return myY;
  }
  public boolean sameSpot(Bug arg)
  {  //code not shown  }
  public void walkTowards(Bug arg,
                        double percent)
  {  //code not shown  }
}
```

1. Mark the private fields "F", the constructors "C", the accessor methods "A", and the instance methods "I".

2. Do the Bug constructors use an explicit `super` command? Y/N Why or why not?

3. The job of the *constructor* is to *initialize* the *private data*, meaning, to assign *values* to the *fields*. Show where that happens in this code.

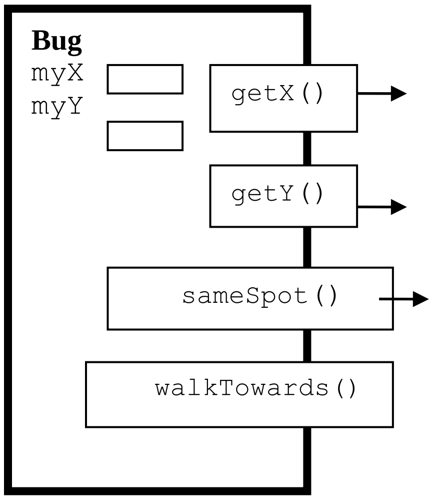
4. Write the code to instantiate a Bug object named `lady` at position (0, 0).

5. Write the code to instantiate a Bug object named `june` at position (37, 145).

6. Tell the `june` Bug to store its x-position in the variable `temp`.

7. Tell a bug named `assassin` to walk halfway towards a bug named `bed`.

It is helpful to picture the Bug class in an *object diagram*:



6. Why is the Bug class pictured inside a thick black box?

7. Why are `myX` and `myY` shown entirely inside the box, but `getX()` and `getY()` extend outside the box?

8. What is the advantage of making fields private, not public?

9. *Modifier methods* change data in an object's fields. The name of a modifier method usually begins with "set". A modifier method also needs an argument. Write the code for a modifier method that modifies `myX`

Discussion
The Turtle class

In the early 1970s M.I.T. introduced a turtle to the LOGO language. Turtles are similar to Bugs and Robots, but are much more complex and powerful. What can Turtles do? Let's look at its API.

1. "static" means that the class owns one copy of each method. Circle the static methods. One of those static methods controls the speed at which the turtles move. Write the command to make the turtles move at speed 5.

____

Hint: recall that the Turtle class "owns" the method.

2. Another class method turns the crawling off, which makes the turtles draw very fast indeed. Write that command here: ____________________________

3. A third class method clears the screen and changes the background color. Change the background to white:

____

4. The "non-static" methods are duplicated inside each turtle object. Star the four methods that actually control the turtle's movements. Tell the SquareTurtle `smidge` to turn right 45 degrees: _________________________

5. Tell `smidge` to leave a line as it walks forward 50 steps: _______________________________

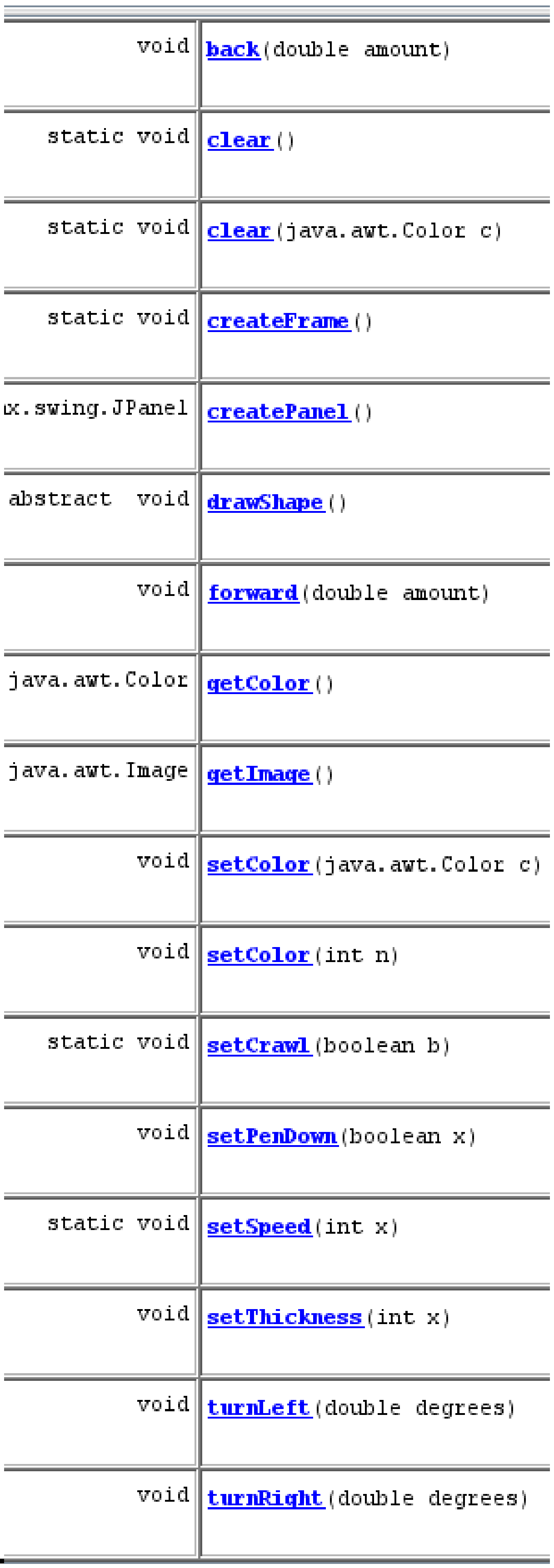
____

6. *Accessor methods* get data from the object's fields. They usually begin with "get". How many accessor methods are in each turtle object? ____

7. *Modifier methods* change data in an object's fields. They usually begin with "set". Tell `smidge` to set its color to yellow and its marker thickness to 10 pixels wide. ____________________________

____

8. Which method is `abstract`? _______________
Since one method is abstract, the whole Turtle class is abstract, meaning that you can't instantiate any Turtle objects directly. Instead, you will extend the Turtle class into concrete subclasses, such as SquareTurtle, PolygonTurtle, and TwistyTurtle. In those subclasses, your job will be to write appropriate code in the `drawShape` method. What abstract classes have you implemented in previous labs? ____________________



void	back(double amount)
static void	clear()
static void	clear(java.awt.Color c)
static void	createFrame()
x.swing.JPanel	createPanel()
abstract void	drawShape()
void	forward(double amount)
java.awt.Color	getColor()
java.awt.Image	getImage()
void	setColor(java.awt.Color c)
void	setColor(int n)
static void	setCrawl(boolean b)
void	setPenDown(boolean x)
static void	setSpeed(int x)
void	setThickness(int x)
void	turnLeft(double degrees)
void	turnRight(double degrees)

Discussion
Classes, Abstract Classes, and Objects

When you first see a new class, the question you should always ask is: what can this class do to itself?

The Turtle class has both static and dynamic fields and methods. The static fields and methods stay with the class and the dynamic fields and methods are duplicated in each object.

The Turtle class has (at least) four static methods, which the class "knows." Class methods often affect the environment of the objects. The Turtle class has set-methods `clear`, `setSpeed`, and `setCrawl` that can change the data stored in three of the private static fields.

The Turtle class is abstract because `drawShape` is abstract. Turtles do not know what shape to draw. The idea is that only the subclasses know what shape to draw.

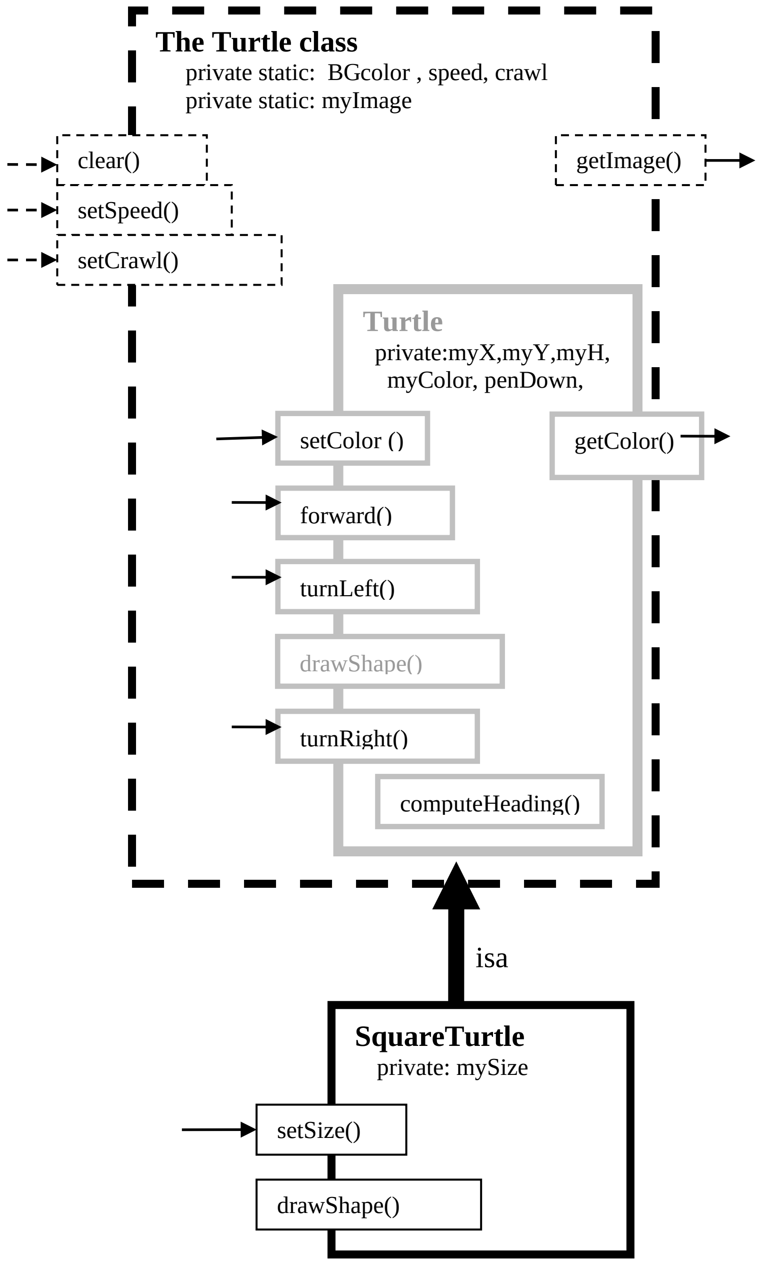
Even though Turtle is abstract, it is still convenient to give it several constructors (which we don't show in these class diagrams). Recall that constructors instantiate the object and initialize the private data. Turtle no doubt has a default constructor. The private fields inherited from Turtle store data about the turtle's position, heading, pen color, and pen state.

Any Turtle subclasses will have constructors with the `super` method which passes initial values to `myX`, `myY`, `myH`, `myColor`, and `penDown`.

Some public methods inherited from Turtle are simple get and set methods, but some are much more complex, such as `turnLeft` and `forward`.

Some private methods are also inherited from Turtle, such as `computeHeading`. There are probably several other private helper methods.

The SquareTurtle class, which inherits lots of functionality from Turtle, has one additional private field `mySize` and one additional set method. It has four constructors. SquareTurtle also implements Turtle's abstract `drawShape` method. SquareTurtles know how to draw squares.



A good assignment would be to write a two-page paper that compares and contrasts the designs of the Bug, Turtle, and Robot resource classes. Include a class diagram similar to that above illustrating the relationship between Robot and Athlete.

Lab06
Square Turtles

Objective
To understand the SquareTurtle class.

Background
Ideally, the API tells you everything you need to know to use SquareTurtle.

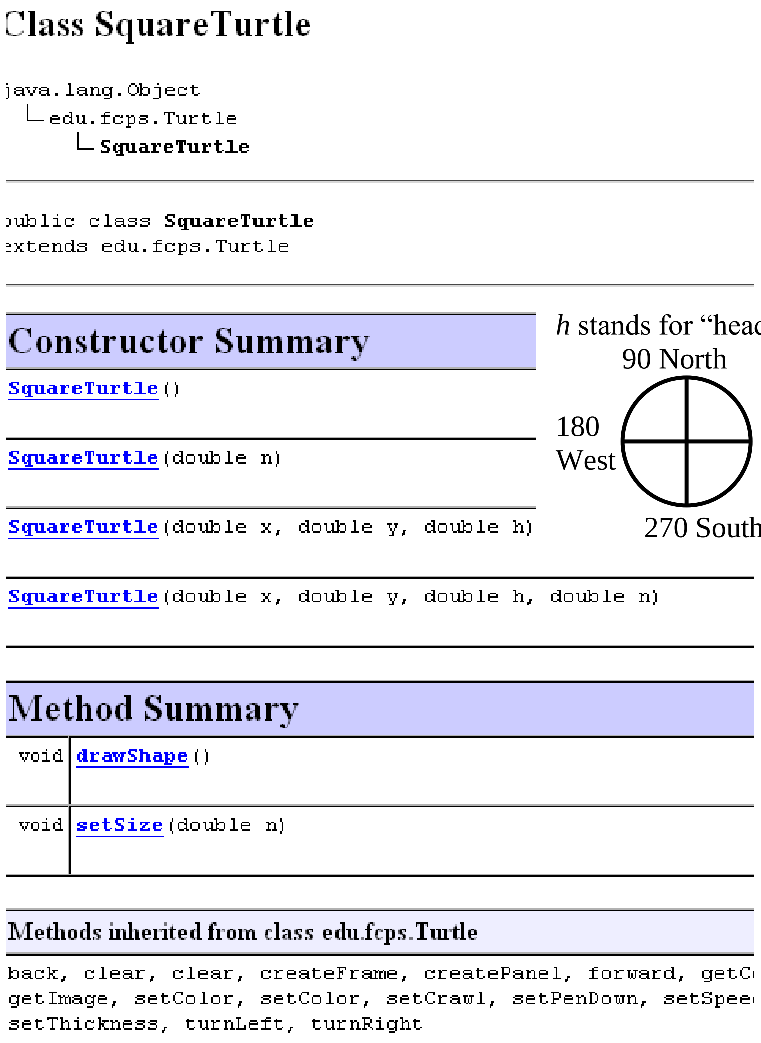
Notice that SquareTurtle extends Turtle.

Notice the four constructors: a default, a one-arg, a three-arg, and a 4-arg constructor. These make it easy to instantiate SquareTurtles in different states, i.e. with different private data. What are *x* and *y*? What is *h*? What is *n*?

Notice the two instance methods. `setSize` is also a modifier method.

Notice that SquareTurtle inherits lots of methods from Turtle.

The API does not tell you that SquareTurtle's default constructor puts a turtle at the center of the screen, at location (300, 300), facing north, with mySize = 50. That is, the programmer made the default constructor equivalent to
`new` `SquareTurtle(300,300,90,50);`



Class SquareTurtle

```
java.lang.Object
  └edu.fcps.Turtle
      └SquareTurtle
```

```
public class SquareTurtle
extends edu.fcps.Turtle
```

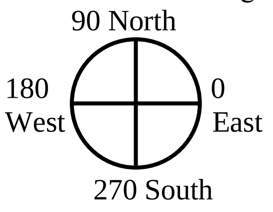
Constructor Summary
SquareTurtle()
SquareTurtle(double n)
SquareTurtle(double x, double y, double h)
SquareTurtle(double x, double y, double h, double n)

Method Summary	
void	drawShape()
void	setSize(double n)

Methods inherited from class edu.fcps.Turtle

back, clear, clear, createFrame, createPanel, forward, getC
getImage, setColor, setColor, setCrawl, setPenDown, setSpee
setThickness, turnLeft, turnRight

h stands for "heading."

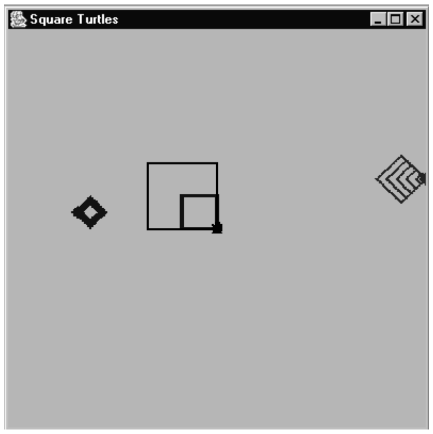


The screen-shot shows three different turtles that were instantiated using three different constructors. Write the calls to those three turtle constructors:

`SquareTurtle left =` ________________________________

`SquareTurtle center =` ______________________________

`Turtle right =` _____________________________________



Specification
Load Filename Unit2\Lab06_7_8_9\SquareTurtle.java. Complete the class so that a SquareTurtle "knows" how to draw a square.

Load Filename Unit2\Lab06_7_8_9\Driver06.java. Create four turtles such that you use all four of the square turtle's constructors. Draw squares of various colors, sizes, and thickness. Change one of your turtle's sizes in the program so that one turtle ends up drawing two squares with different sizes.

Sample Run
`Unit2Lab06.jar`

Extensions
1. Draw the square using a for-loop.
2. Run the program with crawl turned off. What happens to the speed of the program?

Exercises
Lab06

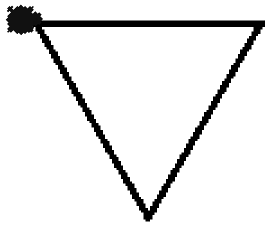
```
 4    import edu.fcps.Turtle;
 5     public class SquareTurtle extends Turtle
 6    {
 7       private double mySize;   //private data
 8        public SquareTurtle()
 9       {
10          super();
11          mySize = 50.0;
12       }
13        public SquareTurtle(double n)
14       {
15          super();
16          mySize = n;
17       }
18        public SquareTurtle(double x, double y, double h)
19       {
20          super(x, y, h);
21          mySize = 50.0;
22       }
23        public SquareTurtle(double x, double y, double h, double n)
24       {
25          super(x, y, h);
26          mySize = n;
27       }
28        public void setSize(double n)
29       {
30          mySize = n;
31       }
32        public void drawShape()
33       {
34         /***********************/
35         /* Your code goes here. */
36         /***********************/
37       }
38     }
```

1. Why is line 4 important?
2. What information is communicated on Line 5?
3. What data does a SquareTurtle "know"?
4. What happens on Line 10?
5. What happens on Line 11?
6. What happens on Line 20?
7. Why is Line 10 different from Line 20?
8. Why are there so many constructors in SquareTurtle?
9. From the programmer's perspective, what must you do in the code of a constructor?
10. On which lines is the code for SquareTurtle's accessor method?
11. On which lines is the code for SquareTurtle's modifier method?
12. Where does Turtle's abstract `drawShape` method become concrete?

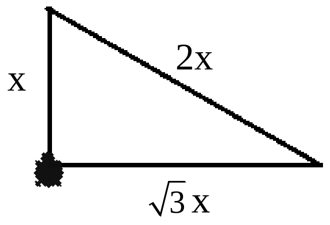
Exercises Continued
Lab06

Equilateral triangles, 30-60-90, and 45-45-90 triangles are important special triangles. Complete the polymorphic `drawShape` method to draw each triangle shown below. The Java method to compute and return the square root is `Math.sqrt`.

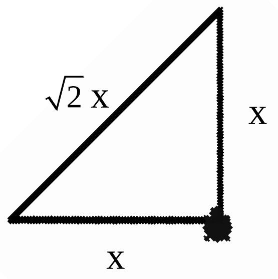
```
1)  public class Equilateral extends Turtle
    {
      private double mySize;
      // constructors, get methods, set methods
      public void drawShape()
      {



      }
```



```
2) public class ThirtySixtyNinety extends Turtle
   {
     private double mySize;
     // constructors, get methods, set methods
     public void drawShape()
     {




   }
```



```
3) public class IsoscelesRight extends Turtle
   {
     private double mySize;
     // constructors, get methods, set methods
     public void drawShape()
     {



     }
```



4) To review: a **class method** is marked by the ___________ keyword, has _________ copy/copies, is called using dot-notation through the ____________, and does ________ need to be instantiated to be used. An example of a class method in the Turtle labs is _______________

5) An **instance method** is marked by the _____________ keyword, has ________ copy/copies, is called using dot-notation through the _____________, and does _________ need to be instantiated to be used. An example of an instance method in the Turtle labs is ___________________

Lab07
Polygon Turtles

Objective
Two private fields in a turtle.

Background
The code for SquareTurtle looks much like that for PolygonTurtle.

```
 4 import edu.fcps.Turtle;
 5 public class PolygonTurtle extends Turtle
 6 {
 7   private double mySize;
 8   private int mySides;
 9   public PolygonTurtle()
10   {
11     super();
12     mySize = 50.0;
13     mySides = 6;
14   }
15   public PolygonTurtle(double n, int s)
16   {
17     mySize = n;
18     mySides = s;
19   }
20   public PolygonTurtle(double x, double y,
                        double h, double n, int s)
21   {
22     super(x, y, h);
23     mySize = n;
24     mySides = s;
25   }
```

What are the private fields in a Polygon Turtle? __________ and ____________

What modifier methods will probably be in Polygon Turtle? _________and _________

On Line 11, what does `super` do?

____

(You should know that the default `super` does not have to be explicitly written. If it is absent, Java automatically calls the default `super`. Line 16 has this hidden call to the default `super`.)

On Lines 12 and 13, what happens to 50.0 and 6? ________________________

On Lines 17 and 18, what happens to n and s? ________________________

On Line 22, what happens to x, y, and h? ____________________________

On Lines 23 and 24, what happens to n and s? ________________________

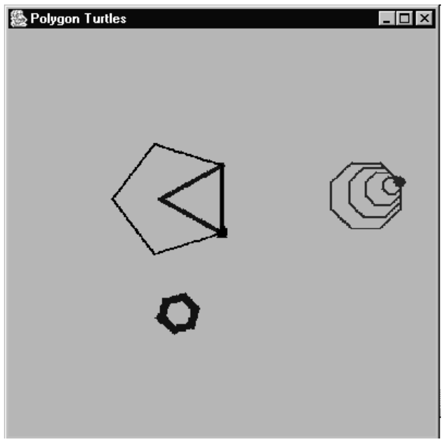
Let's do an example. Given the code

```
      PolygonTurtle odysseus = new PolygonTurtle(300, 0, 225, 100, 4);
      odysseus.drawShape();
```

Where does odysseus start? ____________________   What path will odysseus follow?

Specification
Load Unit2\Lab06_7_8_9\PolygonTurtle.java. Complete the implementation of the PolygonTurtle class. Do you remember the exterior angle theorem from geometry?

Load Unit2\Lab06_7_8_9\Driver07.java. Create different turtles using all of PolygonTurtle's constructors. Then draw 3-, 4-, 5-, 6-, 8, and 10-sided polygons.



Sample Run
`Unit2Lab07.jar`

Exercises

Lab07

1) Complete Bug's *accessor method* below:

```
public int getX()
{


}
```

2) Complete Turtle's *modifier method*:

```
private int mySize;
public void setSize (int n)
{


}
```

3) In Lab07, we drew different-sided polygons by changing the number of sides using `setSides`. It would be nicer to draw different polygons by passing an argument to `drawShape`. For example, `drawShape(4)` would draw a square and `drawShape(5)` would draw a pentagon. Note: Java considers `drawShape()` and `drawShape(int s)` to be different methods. Java does not get confused. We say that the `drawShape` method has been *overloaded.* Write the two methods below. The overloaded `drawShape` method simply sets the number of sides before it draws the polygon.

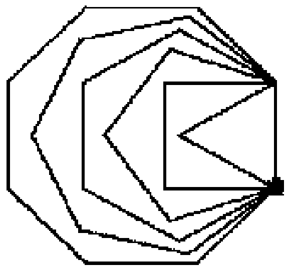
Write the original `drawShape` method:

```
public void drawShape()
{
```

Write the overloaded `drawShape` method:

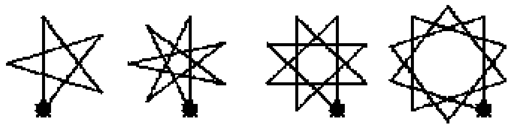
```
public void drawShape(int s)
{
```

4) Write the code to produce this shape. It should use the overloaded `drawShape` from above. Use a for-loop to increment the number of sides.



5) Why does an object-oriented program go to all the trouble to create private data, and then create get and set methods? What advantage is there to having private data and public get and set methods?

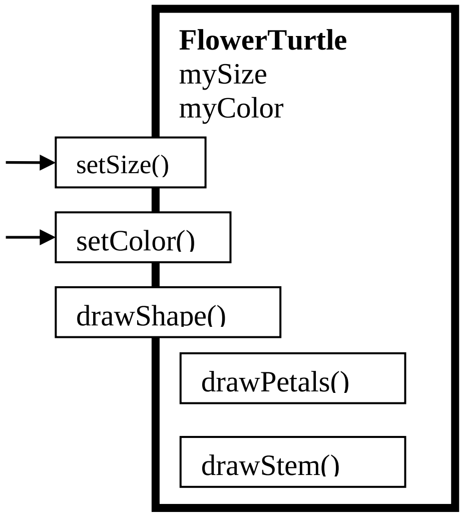
6) Star challenges: the `drawShape` method, with one modification, can draw stars. The same `star` method can draw 5-, 7-, 8- and 10-pointed stars. Can you write it?



Discussion
Access Modifiers

We have previously described `private` fields and `public` accessor and modifier methods. FlowerTurtles have two private fields:

```
public class FlowerTurtle extends Turtle
{
   private double mySize;
   private Color myColor;
   . . .
}
```



In the case of FlowerTurtles, the programmer decided to write two modifier methods, but no accessor methods. The user is able to change `mySize` and `myColor` only through the public modifier methods.

The programmer also decided to make some of FlowerTurtle's methods `private` and some `public`. If a method is `private`, then only methods inside the class can call it. If a method is `public`, then classes outside the class can call it.

A FlowerTurtle's public `drawShape` method draws a flower. "Drawing a flower" is broken into two parts, first the petals and then the stem. This makes the code in `drawShape` as simple as possible:

```
public void drawShape()
{
   drawPetals();                    //calls a private instance method
   drawStem();                      //calls a private instance method
}
```

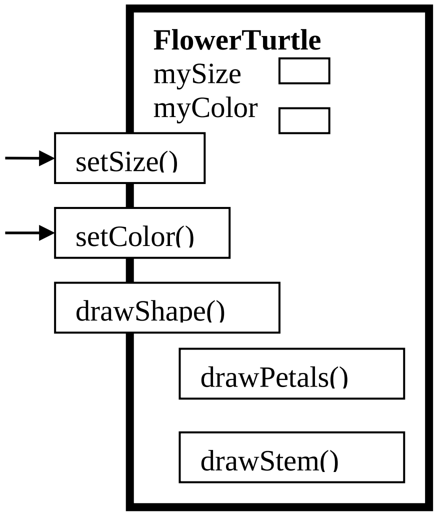
The actual drawing of the flower is accomplished by the two *private helper methods* `drawPetals` and `drawStem`. The two helper methods are `private` because they are only used inside the FlowerTurtle object. They cannot be called from outside the FlowerTurtle object. That's what `private` means.

```
private void drawPetals()
{
   . . .                            //code for private instance method
}
```

Study this driver class:

```
public class Driver08
{
   public static void main(String[] args)
   {
      FlowerTurtle lisa = new FlowerTurtle(300.0, 50.0, Color.RED);
      lisa.setSize(25.0);           //calls a modifier method
      lisa.setColor(Color.BLUE);    //calls a modifier method
      lisa.drawShape();             //calls a public instance method
      lisa.drawPetals();           //error—drawPetals is private!
   }
}
```

Lab08
Flower Turtles



Objective
To tiptoe through the tulips.

Background
One other access issue has to do with overridden methods. Once a method has been overridden, the original superclass method cannot be invoked. Usually, this overriding is exactly what we want. Here is an example where overriding doesn't help us:

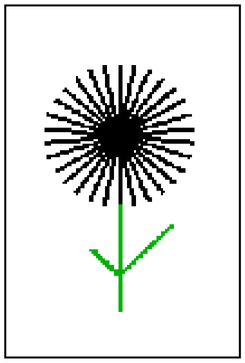
```
public class Driver08
{
   public static void main(String[] args)
   {
      FlowerTurtle lisa = new FlowerTurtle(300.0, 50.0, Color.RED);
      lisa.setSize(25.0);           //calls lisa's setSize()
      lisa.setColor(Color.BLUE);    //calls lisa's setColor()
      lisa.drawShape();
   }
}
```

Lisa's `setSize` and `drawShape` work together to draw flowers of different sizes. Unfortunately, the `setColor` in this class does not make use of the Turtle's `setColor`, which would actually set the color of lisa's pen. However, all is not lost. The superclass's `setColor` can be called from within the subclass class. So from inside FlowerTurtle, Turtle's `setColor` can be invoked, and that setColor will make use of lisa's `myColor`. For example:

```
private void drawPetals()
{
   super.setColor(myColor);  //calls Turtle's setColor, changing the color of the pen
   . . .
}
```

This syntax is only valid from within FlowerTurtle's definition. You cannot do `super.setColor` from outside the subclass of Turtle.

Specification
Load Filename Unit2\ Lab06_7_8_9\FlowerTurtle.java. Complete the FlowerTurtle class by implementing the `drawPetals` and `drawStem` methods. There are thirty petals in the flower shown here. The stem is one-and-a-half times as long as the petals, with leaves at forty-five degree angles. One leaf is equals the petal length, but the other leaf is half the petal length.



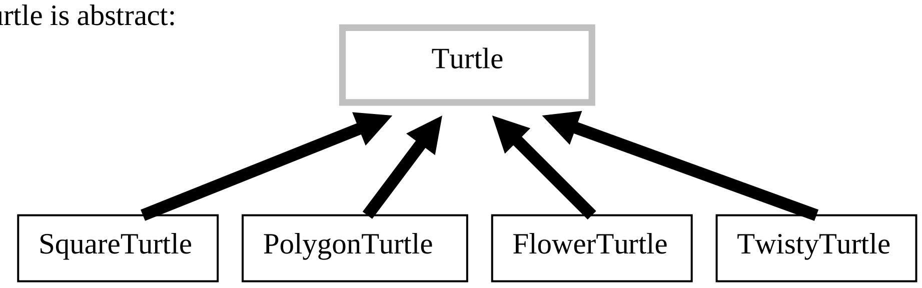
Load Filename Unit2\Lab06_7_8_9\Driver08.java. Draw four pretty flowers in a row, each with different color petals and green stems and leaves. Use a white background.

Sample Run
`Unit2Lab08.jar`

Extension
Draw a garden of randomly-placed flowers of random size. Use a loop. See `Unit2Lab08ext.jar`

Discussion
Polymorphic Behavior in Turtles

Any time you have a hierarchy, you have the possibility of polymorphic behavior. Here is the Turtle hierarchy. Notice that Turtle is abstract:



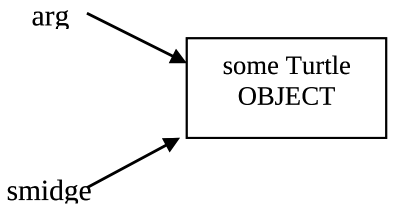
Here is a class method that accepts a Turtle argument:

```
public static void twisties(Turtle arg)
{
   arg.setPenDown(false);
   arg.turnRight((int)(Math.random() * 360));
   arg.forward((int)(Math.random() * 200));
   arg.setPenDown(true);
   arg.drawShape();
}
```

Draw the possible output for each driver method:

```
public static void main(String[] args)
{
   Turtle smidge = new SquareTurtle();
   twisties(smidge);
}

public static void main(String[] args)
{
   Turtle smidge = new PolygonTurtle();
   twisties(smidge);
}

public static void main(String[] args)
{
   Turtle smidge = new FlowerTurtle();
   twisties(smidge);
}
```

Clearly, we have a superclass reference to a subclass object. Which code, the reference's or the object's, is executed by `arg.drawShape`?

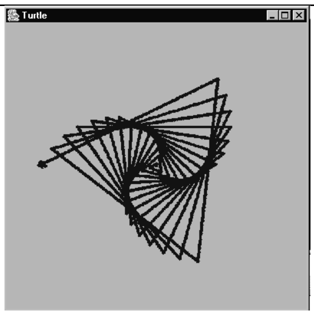
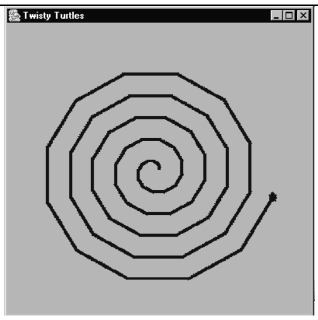
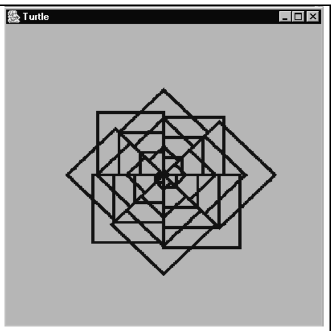


How is `twisties` an example of polymorphism?

Lab09
Twisty Turtles

Objective
To draw twisty designs.

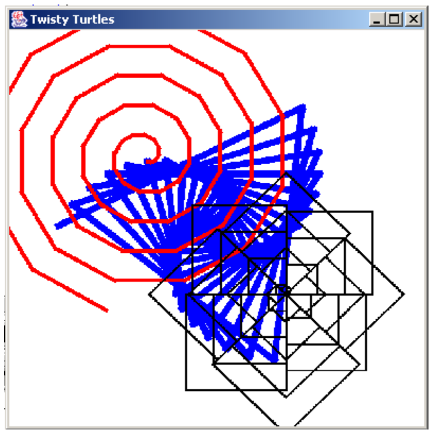
Background
A TwistyTurtle is a subclass of Turtle. A TwistyTurtle does not store any private data. A TwistyTurtle has only a default constructor, which you don't even have to write, because Java generates one for you (under certain conditions). A TwistyTurtle implements only one method, `drawShape`, which has one loop. The loop specifies the beginning length of the line, the end length, and the increment. Each twisty specifies its own turning angle, line thickness, and line color.

	 `TwistyTurtle`	 `TwistyTurtle2`	 `TwistyTurtle3`
begin length	5	0	25
end length	400	150	150
increment	10	2	5
angle	123	30	rotate a square by 45

Experiment with your own values for the length of the side, the increment, and the angle.

Specification
Create Unit2\Lab06_7_8_9\TwistyTurtle.java, TwistyTurtle2.java, and TwistyTurtle3.java. Make three different TwistyTurtles, each implementing a different twisty design in `drawShape`.

Create Unit2\Lab06_7_8_9\Driver09.java. Draw all three twisties on the same screen. Use the class method `twisties` which is given on the previous page.



Sample Run
`Unit2Lab09.jar`

Extension:
Using the class method `twisties`, draw a square, an octagon, a flower, and a twisty. Explain how this is an example of polymorphism.

Extension 2:
Have your original TwistyTurtle class implement the Runnable interface (Unit1, Lab14). All `run` will do is call `drawShape`. In your driver create three twisty turtle objects each with their own thread. Start the threads and watch your turtles twist in parallel. See `Unit2Lab09ext.jar`

Exercises
Writing Your Own Class

Step 1
Choose a class from the list below. Circle it.

Dog	House	Car	City	Book	Boy
Hat	Child	Ring	School	Family	Girl
Cat	Song	Cookie	Flower	Bird	____________

Step 2
Circle the appropriate abstract superclass from the list below:

Animal Vegetable Mineral Idea ________________

Step 3

Design the class. What should your class know? What should your class do? Include at least 1 private field, 2 constructors (one must be the default constructor), and accessors and modifiers (set methods and get methods) for each private field. Design at least 1 appropriate instance method.

Extra credit: Include a class method.

The class outline below may help you. Looking at the code for Bug and SquareTurtle might help you with syntax.

name:

(class method:)

fields:

constructors
 default:

 with arguments:

accessors:

modifiers:

instance method(s):

Step 4

Write the resource class.

```
public class __________ extends ____________
 {
```

Step 5 – Turn this page in to the teacher.

Exercises, continued
Writing Your Own Class

Step 6
Exchange your resource class with another student. Write a driver class (a client class) that uses the other student's resource class. Instantiate two objects using the two constructors. Call at least one set (modifier) method. Call the instance method (and the extra credit class method, if they wrote one.) Using the get (accessor) methods, output the information (`System.out.println()`) stored in each object, with appropriate labels.

```
public class __________
{
     public static void main (String [] args)
     {
```

Lab10
Polka Dots

Objective
To set up a bufferedImage (see Lab03) and a timer. The Polkadot class.

Background
Lines 4-7: we need to import many packages.

This class "isa" JPanel.

Lines 11 & 12: two private constants, both `static` and `final`. Why?

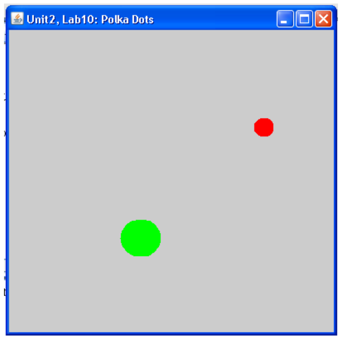
Lines 14-18: six private fields. Line 17 says we will be using a Polkadot object. What is that?! At this point, we don’t need to know!

Lines 19-28: the constructor instantiates objects. It only runs once. As with Turtles, we use an off-screen buffer, `paintComponent`, a Timer, and a Listener to produce motion. The Timer has a loop that calls the Listener every 1000 ms.

All the animation happens in the Listener (lines 33-43). In this code, the `pd` jumps and draws itself. `repaint` on Line 42 actually pastes the updated bufferedImage on the Graphics panel. You need call `repaint` only once at the end of the method.

```
 4 import javax.swing.*;
 5 import java.awt.*;
 6 import java.awt.event.*;
 7 import java.awt.image.*;
 8 public class PolkaDotPanel extends JPanel
 9 {
10   //constants
11   private static final int FRAME = 400;
12   private static final Color BACKGROUND = new Color(204, 204, 204);
13   //fields
14   private BufferedImage myImage;
15   private Graphics myBuffer;
16   private Timer t;
17   private Polkadot pd;
18   private int xPos, yPos;
19   public PolkaDotPanel()
20   {
21     myImage =  new BufferedImage(FRAME, FRAME, BufferedImage.TYPE_INT_RGB);
22     myBuffer = myImage.getGraphics();
23     myBuffer.setColor(BACKGROUND);
24     myBuffer.fillRect(0, 0, FRAME, FRAME);
25     pd = new Polkadot();
26     t = new Timer(1000, new Listener());
27     t.start();
28   }
29   public void paintComponent(Graphics g)
30   {
31     g.drawImage(myImage, 0, 0, getWidth(), getHeight(), null);
32   }
33   private class Listener implements ActionListener
34   {
35     public void actionPerformed(ActionEvent e)
36     {
37      /**************************
38         your code goes here
39      **************************/
40       pd.jump(FRAME, FRAME);
41       pd.draw(myBuffer);
42       repaint();
43     }
44   }
45 }
```

Specification
Load Unit2\Lab10\PolkaDotPanel.java. Study the code above!
Load Unit2\Lab10\PolkaDot.java. Complete this resource class; see the next page.
Load Filename Unit2\Lab10\Driver10.java. Compile and run. You will see that polkadots keep appearing, one every second. First, add some code at lines 37-39 in PolkadotPanel so only one polkadot appears to jump around the screen. After that works, make two polkadots of different sizes and colors jump around the screen.

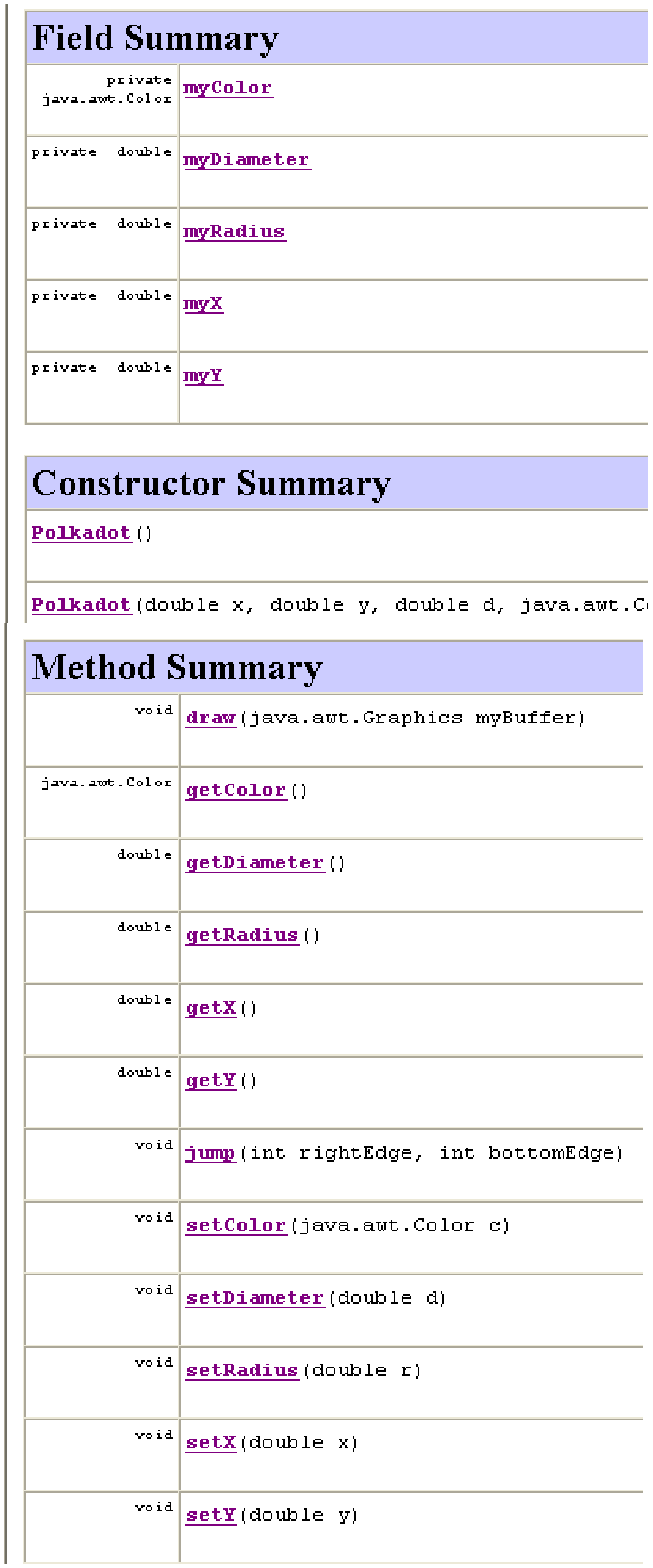


Warning: A common error is to write Line 25 as `Polkadot pd= new Polkadot();` Write Line 25 that way. Then explain what the error message `null pointer exception` means.

Sample Run `Unit2Lab10.jar`

Discussion
Polkadot

What can a polkadot do? What do they know? The API below is generated from the code at the right.



Field Summary

private java.awt.Color	myColor
private double	myDiameter
private double	myRadius
private double	myX
private double	myY

Constructor Summary

Polkadot()
Polkadot(double x, double y, double d, java.awt.C

Method Summary

void	draw(java.awt.Graphics myBuffer)
java.awt.Color	getColor()
double	getDiameter()
double	getRadius()
double	getX()
double	getY()
void	jump(int rightEdge, int bottomEdge)
void	setColor(java.awt.Color c)
void	setDiameter(double d)
void	setRadius(double r)
void	setX(double x)
void	setY(double y)

```
import java.awt.*;
public class Polkadot
{
  private double myX;    // x coordinate of center
  private double myY;    // y coordinate of center
  private double myDiameter;
  private Color myColor;
  private double myRadius;
  public Polkadot()
   {
     myX = 200;
     myY = 200;
     myDiameter = 25;
     myColor = Color.RED;
     myRadius = myDiameter/2;
   }
   public Polkadot(double x, double y,
                              double d, Color c)
   {
     myX = ___;
     myY = ____;
     myDiameter = ____;
     myColor = ____;
     myRadius = _____;
   }
   public double getX()
   {
     return myX;
   }
   public double getY()
   {
      _____________
   }
   public double getDiameter()
   {
      _____________
   }
   public Color getColor()
   {
     return myColor;
   }
   public double getRadius()
   {
     return myRadius;
   }
   public void setX(double x)
   {
     myX = x;
   }
   public void setY(_____________)
   {
     ______________
   }
   public void setColor(Color c)
   {
     myColor = c;
   }
   public void setDiameter(double d)
   {
     myDiameter = d;
     myRadius = d/2;
   }
   public void setRadius(double r)
   {
      __________________
      __________________
   }
   public void jump(int rightEdge,int bottomEdge)
   { //code
   }
   public void draw(Graphics myBuffer)
   {  // a polkadot draws itself from its center
```

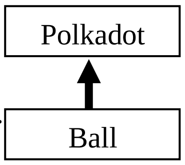
Discussion
Ball

```
 2 import java.awt.*;
 3 public class Ball extends Polkadot
 4 {
 5  private double dx;
 6  private double dy;
 7  // constructors
 8  public Ball()          //default constructor
 9   {
10    super(200, 200, 50, Color.BLACK);
11    dx = Math.random() * 12 - 6;          // horizontal
12    dy = Math.random() * 12 - 6;          // vertical
13   }
14  public Ball(double x, double y, double dia, Color c)
15   {
16    super(x, y, dia, c);
17    dx = Math.random() * 12 - 6;
18    dy = Math.random() * 12 - 6;
19   }
20  //modifier methods
21  public void setdx(double x)
22   {
23    dx = x;
24   }
25  public void setdy(double y)
26   {
27    dy = y;
28   }
30  //accessor methods
31  public double getdx()
32   {
33    return dx;
34   }
35  public double getdy()
36   {
37    return dy;
38   }
40  //instance methods
41  public void move(double rightEdge, double bottomEdge)
42   {
43       ______________________________
 .  .  .
50   }
```

What should a ball "know"—i.e., what fields? What should a ball be able to do—i.e., what methods? Can a Ball make use of fields and methods in Polkadot?

For its fields, a ball needs to know at least its location, diameter, and color. In fact, there were five *private fields* in Polkadot.
1. What were they?
__________ _________ _________
__________ _________

Because they share so much, it seems reasonable to make Ball extend Polkadot, as shown in this UML diagram.



In Java, Ball inherits the *public methods*, but not the *private fields*, from Polkadot. Ball cannot directly access `myX` and `myY`, or any of the other private fields. But the programmer was thinking ahead, because she wrote public accessor and modifier methods for all of Polkadot's private fields. When Ball needs to access those fields, the Ball class must use the public `getX` and `setX` methods.

2. As you know, a Ball inherits all public instance methods from Polkadot. What do Balls know how to do, because Polkadots know how to do them? ______ _______

3. The programmer gave Ball two new fields on Lines 5 and 6. What are the new fields in Ball? _____ ______
Because these fields are in Ball, the Ball class can access them directly. But we also write public accessor and modifier methods for them, in case we later extend the Ball class.

4. What do you think these new fields do?

5. On what lines are the constructors?

6. What does the keyword `super` do?

7. What are the attributes of the default Ball object?

8. The author also gave Ball one new public method `move`. In words, describe what has to happen to make the ball appear to move slowly horizontally.

9. On Line 43 in the code above, write one line that causes this ball object to move `dx` spaces horizontally.

Lab11
Bouncing Pinball

Objective
The Ball class and motion.

Background
Here is the `actionPerformed` method of the PinballPanel. Just as in the Polkadot lab, the buffer is painted gray and then a new image is repainted. If the ball moves a little bit, and the timer fires quickly, the ball appears to roll around the screen. Notice we have encapsulated both the movement and the painting in the Ball class, i.e., ball objects "know" how to move and to paint themselves.

```
35  private class Listener implements ActionListener
36  {
37    public void actionPerformed(ActionEvent e)
38    {
39      myBuffer.setColor(BACKGROUND);
40      myBuffer.fillRect(0,0,FRAME,FRAME);
41      ball.move(FRAME,FRAME);
42      ball.draw(myBuffer);
43      repaint();
44     }
45   }
```

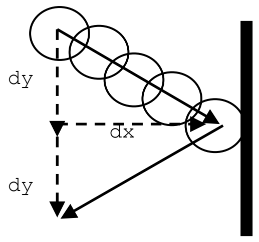
What is a Ball? It makes sense that a Ball should inherit many items that are already in Polkadot. (Like what?) What's new in Ball is the `dx` and `dy`, for delta-x and delta-y, meaning the change in x and the change in y. The Ball class chooses both of these values as random numbers. What if the `dx` is 0? What if the `dy` is 0? What if both are 0?

Lines 7-40: of course, the Ball class has its own constructors, accessors, and modifiers.

```
 2    import java.awt.*;
 3    public class Ball extends Polkadot
 4    {
 5       private double dx;
 6       private double dy;
 . . .
41    //instance methods
42    public void move(double rightEdge, double bottomEdge)
43    {
44      setX(getX()+ dx);                  //move horizontally
45
46      if(getX() >= rightEdge - getRadius())  //hit right edge
47      {
48        setX(rightEdge - getRadius());
49        dx = dx * -1;
50      }
51      /*  more code goes here  */
52
53
```

How can we make a ball move horizontally? Get the ball's x-position, add the delta-x, and make it the new x-position. The code for this is on Line 44.

How does the ball know when it hits the right edge? It continually checks to see if its x-position is too large. One trouble is that the ball knows the coordinates of its center, but it bounces off the edge when the ball's radius hits the edge. The code in Line 46 accounts for that. What does it mean to bounce, that is, how do the dx and dy change when a ball hits the right edge? How many edges do you have? What does it mean, in terms of dx and dy, to bounce off each edge?



Specification
Copy Unit2\Lab10\Polkadot. Since you don't instantiate a polkadot, why do you have to copy it?
Load Filename Unit2\Lab11\Ball.java. The ball moves sideways and bounces off the right edge (don't change it!). Use that code as a model to make the Ball move in all directions and bounce off all edges.
Load Filename Unit2\Lab11\PinballPanel.java.
Create Filename Unit2\Lab11\Driver11.java. Create an appropriate driver.

Sample Run `Unit2Lab11.jar`

Extension Make a new class extending Ball that draws the `tj.jpg` image from Lab02. The image will bounce! Note that TJ has code for exactly one (1) overridden method:

```
public class TJ extends Ball
{  public void draw(Graphics myBuffer)
```

Exercises
Labs10 and 11

Answer the questions about these five (very simplified and rudimentary) classes. A `{…}` in the code below indicates that code has been omitted.

```
public class P
{
  private int myX;
  public P(int x)
  {
    myX = x;
  }
  public int getX()
  {…}
  public void drawMe()
  {…}
}
```

```
public class B extends P
{
  private double dx;
  public B()
  {
    super(100);
    dx = 5;
  }
  public double getdx()
  {…}
}
```

```
public class T extends B
{
  public void drawMe()
  {…}
}
```

```
public class D
{
 private final int xx=10;
 public static void drawAll()
 {









 }
public static void main(String[] args)
 {
   drawAll();
 }
}
```

1. Circle the true statements:
   B isa P   P isa B   A isa P
   T isa B   T isa P   A hasa P
2. Can P directly access myX? Y/N
3. Can B directly access myX? Y/N
4. Can A directly access myX? Y/N
5. If P needs to access myX, it can use _____ or _____
6. If B needs to access myX, it must use _______
7. If T needs to access myX, it must use ________
8. Can P directly access dx? Y/N
9. Can B directly access dx? Y/N
10. Can T directly access dx? Y/N
11. If T needs to access dx, it must use __________
12. Can A directly access xx? Y/N
13. Can T directly access xx? Y/N
14. Can D directly access `p`, `b`, or `t`? Y/N
15. Can `xx` be changed? Y/N
16. Why doesn't P have an explicit `super`? ____
17. Why does B have an explicit `super`? _____________________________________
18. Why doesn't T have a constructor? _____________________________________
19. Why does B have `dx=5` ? _______________________________________
20. Does B have a `drawMe`? Y/N Explain: __________________________________
21. Why does T have its own code for `drawMe`? ____________________________
22. What word describes the relationship between P's `drawMe` and T's `drawMe?` ___________________
23. Class D has a field xx that is `final` What does `final` mean? ______________________
24. Class D has a `static` method `drawAll`. Why does `static` mean? (3 things)

25. D is a _________________. In D, **write code** that instantiates each object (P with a size of 10, B with its default size, and T with its default size) and tell each object to draw itself.

Lab12
Get the Prize

Objective
Using multiple classes with appropriate responsibilities.

Background
Object-oriented programming attempts to distribute appropriate responsibilities to the different classes. Polkadots should do polkadot stuff, balls should do ball stuff, and panels should organize them. In this lab, panel's responsibility is also to decide when the moving ball actually hits the polkadot, to count those times, and to display the count. That is, PrizePanel eventually will have an additional private field to store the number of hits, a private method called `collide` and a private method called `distance`.

Study the panel's Listener. Line 43: the ball moves itself

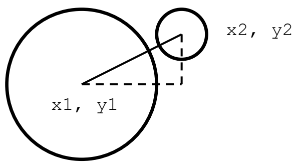
Line 44: the panel calls its private `collide` method, *passing* the ball and the polkadot.

Lines 55-60: The `collide` method then calculates the distance and if the ball hits the polkadot, the hit counter goes up by one and the polkadot jumps to a new location.

Lines 46-51: the ball draws itself, the polkadot draws itself, and the "Count: 1" string is drawn in the upper-right hand corner of the panel.

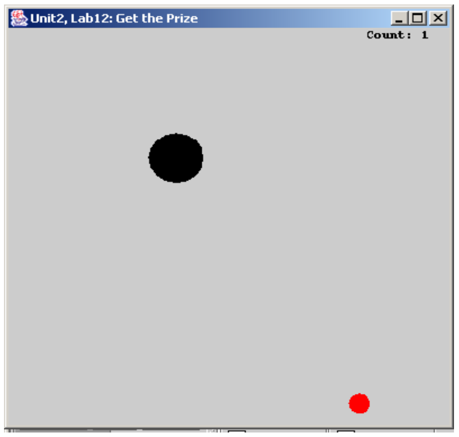
```
36  private class Listener implements ActionListener
37  {
38   public void actionPerformed(ActionEvent e)
39   {
40     myBuffer.setColor(BACKGROUND);
41     myBuffer.fillRect(0,0,FRAME,FRAME);
42
43     ball.move(FRAME, FRAME);
44     collide(ball, pd);
45
46     ball.draw(myBuffer);
47     pd.draw(myBuffer);
48
49     myBuffer.setColor(Color.BLACK);
50     myBuffer.setFont(new Font("Monospaced", Font.BOLD, 24));
51     myBuffer.drawString("Count: " + hits, FRAME - 150, 25);
52     repaint();
53   }
54  }
55  private void collide(Ball b, Polkadot pd)
56   {
57     double d = distance(  /* 4 arguments */  );
58     if( d <=
59
```

Line 58: How do they know they collide? The ball and the polkadot each "know" their own position. If the distance between their centers (recall the distance formula from algebra) is less than the sum of their respective radii, then they hit.



The distance formula uses both `Math.sqrt` and `Math.pow`, which return `doubles`. The expression to square the difference between x2 and x1 is `Math.pow(x2 – x1, 2);`

Specification
Copy Polkadot.java and Ball.java from Lab11.



```
Count: 1
```

Load Filename Unit2\Lab12\PrizePanel.java. The panel’s fields are given to you. Using PinballPanel as a model, complete the constructor and `paintComponent`. Type in the `actionPerformed` method given above. Complete `collide` and `distance`. Then modify PrizePanel to count and report the number of times the ball hits the polkadot.

Filename Unit2\Lab12\Driver12.java. Create an appropriate driver.

Sample Run
`Unit2Lab12.jar`

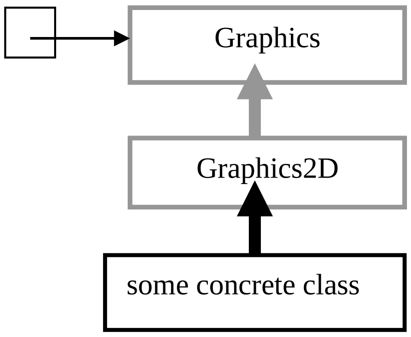
Lab13--Project
Riddle Me This

Objective
To make your teacher and classmates laugh.

Background
Ever since Lab03, we have pretended that `myImage`'s image was a Graphics type object. In truth, Graphics is abstract, so the actual object created is a subclass of Graphics. Then when the Java programmers decided to add more powers to the Graphics class, they chose to extend it. They named the new class Graphics2D, and also kept it abstract. Look at the API for Graphics2D.

Graphics2D inherits familiar Graphics methods like `fillRect`, `drawPolyline`, `setColor`, and `setFont`. It overrides and overloads the method `drawString`. Graphics2D helpfully defines a new method `setStroke`. The method `setStroke` and a `BasicStroke` object, whose constructor takes a *float* (not a double) decimal number as an argument, can be used to change the thickness of lines. Here is an example:

```
myBuffer.setStroke(new BasicStroke(10.0f));    //10 pixels wide
```



So far, so good. However, the method `myImage.getGraphics` returns a Graphics reference, because the original Graphics class was written that way. The question is, how do we make the brand-new and better Graphics2D commands available? We can't, unless we ***cast*** the `getGraphics` return reference to its subclass type. The new reference becomes "bigger" because it points to the subclass with all its methods, including the new ones. We uncover the Graphics2D object that we know are there, but `getGraphics` does not. The *cast* occurs in line 19, inside the parentheses. `myBuffer` now accesses Graphics2D commands.

Clearly we are using the same structure that we used in Lab10 and Lab11. What's new is being able to use `setStroke`.

```
 8 public class ProjectPanel extends JPanel
 9 {
10   private static final ImageIcon THOMAS =
                                  new ImageIcon("tj.jpg");
11   private BufferedImage myImage;
12   private Graphics2D myBuffer;
13   private Timer t1, t2;
14   private int xPos;
15   public ProjectPanel()
16    {
17     xPos = 50;
18     myImage =  new BufferedImage(700, 500,
                            BufferedImage.TYPE_INT_RGB);
19     myBuffer = (Graphics2D)myImage.getGraphics();
20     t1 = new Timer(100, new Listener1());
       . . .
25
```

Specification
Open RiddlePanel.java. Ask, answer, and illustrate a riddle. Use at least one ImageIcon image (refer to Lab02), various polygons, at least two different pen-widths, text output, two timers, and two listeners. One timer and its listener moves an image across the screen. The second timer waits a while, then reveals the answer to the riddle.

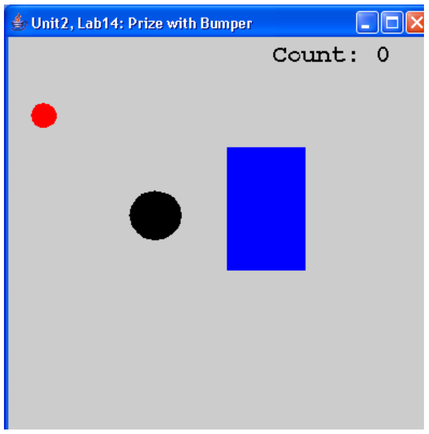
Open Unit2\Lab13\RiddleDriver.java. Compile and run.

Sample Run `Unit2Lab13.jar`

Lab14
Prize with Bumper

Objective
Designing a class from scratch (mostly).

Background
The panel is just like Lab12, with a moving ball, a jumping prize, and a counter, but has a rectangular bumper somewhere in the middle of the screen. The bumper is (of course) an object of the Bumper class. The author had to ask him/herself a series of questions, including:



1. Is there a suitable class that Bumper could extend?

2. What private data should a bumper know about itself?

3. What constructors would be nice to have?

4. What get and set methods are needed?

5. What instance methods are needed? That is, what should a bumper know how to do?

#1: The author thought about extending Polkadot because both have x and y coordinates. However, since Bumper does not have a diameter or a radius, it seemed better not to have Bumper extend anything.

#2: Bumpers at a minimum need to know their upper left hand corner, their width, and their height, because that is how they are drawn. It might be nice to have different colored bumpers.

#3: The default constructor always places the bumper in the same place. It would be nice to place it in different places with different sizes. Let's make two constructors, a no-arg and a 5-arg constructor.

#4: Get-methods are needed to get the x, y, width, and height. Set-methods might not be needed, unless the position, size, and color of the bumper need to be changed. If we make Pong, for instance, we will need `setY`.

#5: A bumper should know how to draw itself, like Polkadot and Ball. If a bumper knew how to jump to a random location, the game would change each time we ran it.

Somewhere, we must define how the ball bounces off the bumper. Bouncing off the corners of the bumper involves "vector algebra." Luckily for you, the author already wrote and encapsulated all that code in a separate class. `BumperCollision.collide(bumper, ball)` is the syntax for calling that magic method.

Specification
Copy Polkadot.java, Ball.java, and Driver12.java from Lab12.

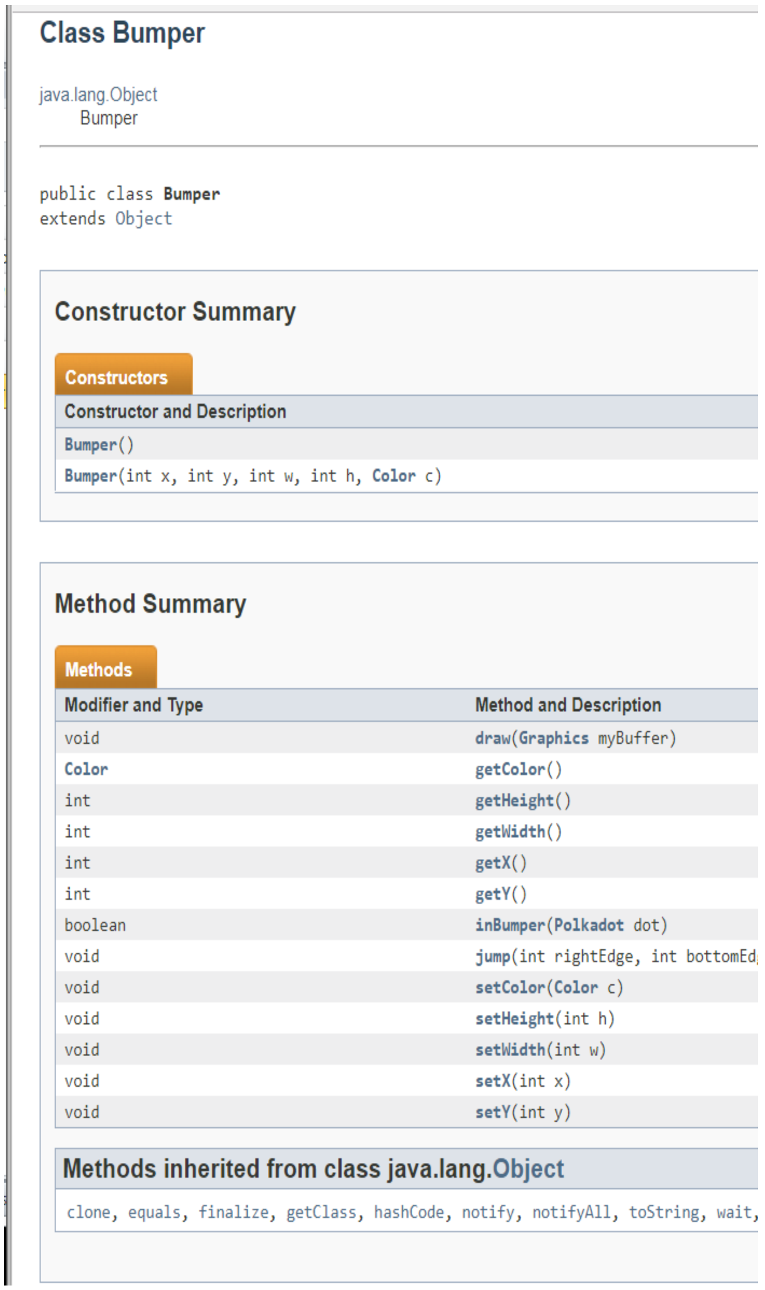
Load Filename Unit2\Lab14\Bumper.java. Complete the design of the class. Create 5 fields, 5 accessor methods, 5 modifier methods, and 2 constructors. You have 1 instance method to complete. See the next page.

Load Filename Unit2\Lab14\BumperPanel.java. Instantiate all three objects, than jump them to random locations.

Sample Run
`Unit2Lab14.jar`

Lab14
Exercises

Study the API below. Write the code for Bumper. Your code must conform to the specifications below. For example, the fields in Bumper are all `int`s. You must spell "Height" correctly. The `inBumper` method tests to see if the polkadot is inside the bumper or not. The `inBumper` method uses the private `distance` method which you saw in Lab12.



Class Bumper

java.lang.Object
    Bumper

```
public class Bumper
extends Object
```

Constructor Summary

Constructors

Constructor and Description
Bumper()
Bumper(int x, int y, int w, int h, Color c)

Method Summary

Methods

Modifier and Type	Method and Description
void	draw(Graphics myBuffer)
Color	getColor()
int	getHeight()
int	getWidth()
int	getX()
int	getY()
boolean	inBumper(Polkadot dot)
void	jump(int rightEdge, int bottomEd
void	setColor(Color c)
void	setHeight(int h)
void	setWidth(int w)
void	setX(int x)
void	setY(int y)

Methods inherited from class java.lang.Object

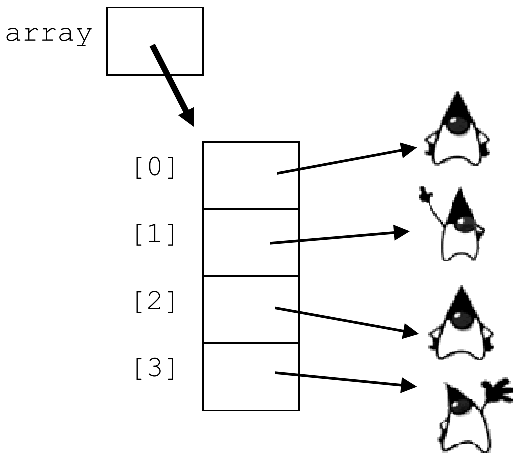
clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait,

```
public class Bumper
 {
  //private fields







  //constructors
 public Bumper()
   {






   }
 public Bumper(int x, int y,
               int w, int h, Color c)
   {






   }
 // accessor methods














  // modifier methods








. . .


  //instance methods
  public void jump(int rightEdge, int
                          bottomEdge)
```

Lab15
Karel the Robot

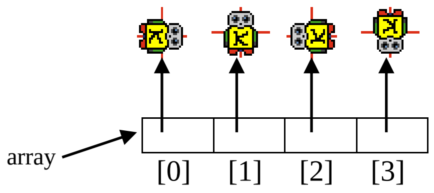
Objective
To simulate an animated gif.

Background
We accomplished the appearance of moving balls by quickly erasing a picture and drawing it again in a slightly new position. The same appearance of motion can be accomplished by cycling through a series of images stored in an array, if the images are chosen carefully. (This is the way animated gifs and sprites work.)



As the values of the array's index cycles from 0 to 1 to 2 to 3 to 0 to 1 and so on, the image that actually gets displayed keeps changing. Each image is just a still shot but the cycling of images gives the appearance of motion.

This lab will use karel the robot images stored in an array. Karel can face either east, north, west, or south. Your code will have to select which image is appropriate. The first, basic assignment is to make the robot move around the edges of the grid.



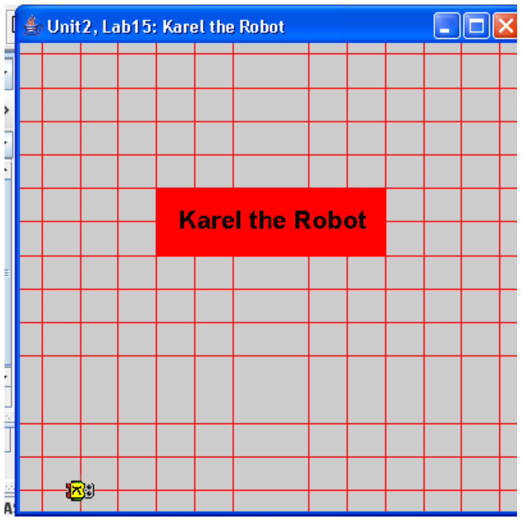
As with all images, the (xPos, yPos) coordinates of the images refer to the upper-left hand corner. Unfortunately, the karel images are not square. That makes proper spacing a bit tricky in this lab. To get the width and height of the image stored in `myArray[0]` use the commands:

```
int width = myArray[0].getImage().getWidth(null);
int height = myArray[0].getImage().getHeight(null);
```

To actually display the image stored in `myArray[0]` use the command (last seen in Lab02):

```
myBuffer.drawImage(myArray[0].getImage(), xPos, yPos, null);
```

Specification
Load Filename Unit2\Lab15\KarelPanel.java. You will have to draw the gridlines. Make Karel travel around the grid, turning so its head is always pointing forward. Then make Karel do a dance.



Load Filename Unit2\Lab15\Driver15.java.

Sample Run
`Unit2Lab15.jar`

Lab16
Mouse Input

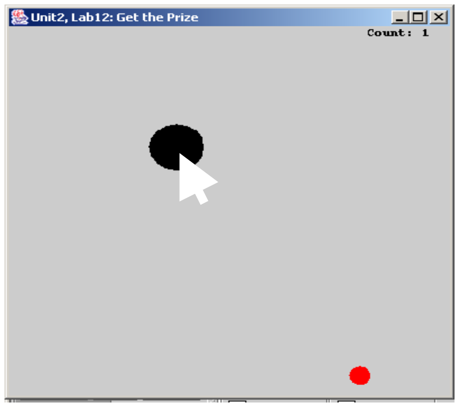
Objective
To modify the “Get the Prize” lab (Lab 12) to include mouse input.

Background
Not all listeners are ActionListeners registered with Timers. You can also register a MouseListener with the JPanel itself, as shown on Line 30.

```
16 public PrizePanel()
17  {
  . . .
30   addMouseListener(new Mouse());
31   t = new Timer(10, new Listener());
32   t.start();
33  }
. . .
37  private class Mouse extends MouseAdapter
38  {
39    public void mousePressed( MouseEvent e )
40    {
41      ball.setX( e.getX() );
42      ball.setY( e.getY() );
43    }
44  }
```

The Listener object on Line 31 is the same Listener from Lab12, which implements the ActionListener interface, which controls the motion of the objects.

On line 37, the Mouse class could either implement the MouseListener interface or extend the MouseAdapter class. If you implement MouseListener you'll need to define five methods, even if you only use one of them. To avoid this complication, the MouseAdapter class provides default (empty) definitions for these methods, so only the methods that you override need to be defined.



Notice the mysterious `e.getX()` method. It looks like it could be a standard accessor method on the `e` object. This object `e` is instantiated somewhere up in the MouseAdapter hierarchy. Whenever you click, this `e` object is automatically passed to the `mousePressed` method. The `e` object has its own set of methods. It turns out that `e.getX()` returns the x location of the mouse pointer, and similarly for `e.getY()`. Study Lines 41 and 42, which move the ball to the spot where you clicked.

Specification
Copy all files from Lab12. Set the JFrame's size to (408, 438), which makes the mouse's location in the JFrame correspond to its location in the JPanel. Then modify the `mousePressed` method to get three different behaviors from e:

button	boolean method	desired behaviors
left click	<none>	moves the polkadot to where you clicked
right click	`e.isMetaDown()`	moves the ball to where you clicked
Shift + left click	`e.isShiftDown()`	randomly changes the speed and direction of the ball

Sample Run
`Unit2Lab16.jar`

Extension
Implement an interesting behavior that is allowed by the MouseMotionListener interface. MouseMotionListener objects are registered with the addMouseMotionListener method.

Lab17
Keyboard Input

Objective
To modify the “Get the Prize” lab (Lab 12) to include keyboard input.

Background
Still another Listener is a KeyListener, which can be registered with any JPanel. Look at Line 30 for an example.

Now each application can have three listeners, one each for the mouse, the keyboard, and the timer.

The private Key class on Line 60 could either implement the KeyListener interface or extend the KeyAdapter class. If you implement KeyListener you’ll need to define three methods, even if you only use one of them. On the other hand, the KeyAdapter class provides default (empty) definitions for these methods, so only the methods that you actually use need to be defined.

Notice that Lines 64, 66, 68, and 70 call accessor methods of a mysterious KeyEvent object named `e`, so that `e.getKeyCode()` returns the key that was pressed. VK_W, etc., are constants in the KeyEvent class. The result is that Lines 64 to 71 move the ball either 1 pixel up, down, left, or right.

```
  4 //imports
    . . .
  9 public class PrizePanel extends JPanel
 10 {
      . . . //fields
 16   public PrizePanel()   //constructor
 17   {
        . . .
 30     addKeyListener(new Key());
 31     setFocusable(true);
        . . .
 44   }
 45   //private listener classes
      . . .
 60   private class Key extends KeyAdapter
 61   {
 62     public void keyPressed(KeyEvent e)
 63     {
 64      if(e.getKeyCode() == KeyEvent.VK_W)
 65          ball.setY( ball.getY()-1 );
 66      if(e.getKeyCode() == KeyEvent.VK_Z)
 67          ball.setY( ball.getY()+1 );
 68      if(e.getKeyCode() == KeyEvent.VK_A)
 69          ball.setX( ball.getX()-1 );
 70      if(e.getKeyCode() == KeyEvent.VK_S)
 71          ball.setX( ball.getX()+1 );
 72     }
 73   }
 74   //public instance methods
      . . .
150 }
```

Specification
Copy all the resource files from Lab16. Use the code above as a model so that the **arrow keys** move the **polkadot ten pixels** at a time. Do not allow the keys to move the polkadot outside the boundaries of the panel.

Load Filename Unit2\Lab17\Driver17.java. Notice that the driver calls the panel's method `requestFocus`. Alternatively, you can change the focus by clicking on the panel, which makes the keys work on that panel.

Sample Run
`Unit2Lab17.jar`

Extensions
1) Make a new folder for Pong. Copy into it all needed files from Lab14. Modify BumperPanel so that the keys move the bumper vertically.
2) Make a two-person Pong game. Keep score.
3) In Lab15, modify KarelPanel to use both mouse and keyboard input.
4) A website that students have found helpful is http://www.faqs.org/docs/javap/index.html

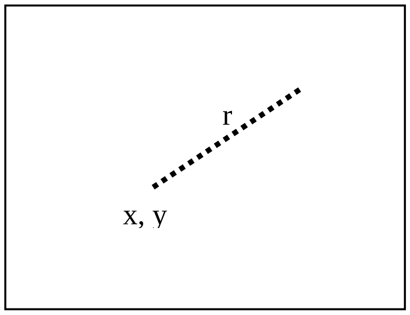
Discussion
Turtle, from scratch

The shell shown below begins to code the Turtle class described in Lab06, but without a Turtle image or animation. Why are the `img` and the colors all static fields? Why are `x`, `y`, and `theta` **not** static fields?

```java
import java.awt.*;
import java.awt.image.*;
public class Turtle
{
     private static BufferedImage img;
     private static int black = 0;
     private static int blue = 255;         // 2^8-1
     private static int green = 65280;      //(2^8-1)*2^8
     private static int red = 16711680;     //(2^8-1)*2^16
     private static int white = 16777215;  // 2^24-1
     private double x, y, theta;          // what does a Turtle know?
     static                               // a static initializer list
     {
          img = new BufferedImage(800, 600, BufferedImage.TYPE_INT_RGB);
     }
     public Turtle()
     {
          x = img.getWidth()/2;             //start at the center
          y = img.getHeight()/2;            //start at the center
          theta = 90;                       //90 degrees faces north
     }
     public static Image getImage()
     {
          return img;
     }
     public void turnLeft(double a)
     {
          theta += a;
     }
     public void forward(double r)
     {
          // Your code goes here!
     }
}
```

You'll need to complete the method `forward` by calculating **each** pixel on the path and setting it to white. For instance, to set **one** pixel at location (x, y) of `img` to black, the command is:

```java
     img.setRGB( (int)x, (int)y, black );
```



You'll need a for-loop, obviously. You might find the trig concepts from Lab03 to be useful. Recall that Java works in radians. If you try to set a pixel that is outside the area of image, you will get an error message.

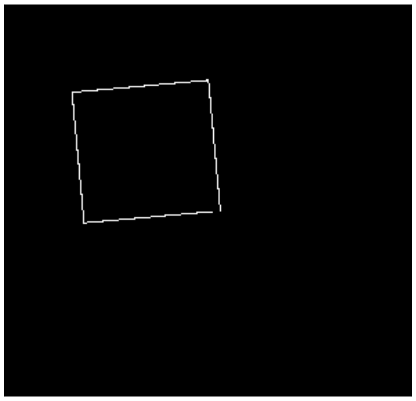
Add more constructors and methods to your Turtle class. Some possible methods to add are `turnRight`, `back`, `penDown`, `setColor`, and `clear`.

Lab18
Turtle, from scratch

Objective
To solve a problem by first writing a resource class.

Background

```
import java.awt.*;
import javax.swing.*;
public class Lab18 extends JPanel
{
     public static void main(String[] args)
     {
          Turtle t = new Turtle();
          t.turnLeft(5);
          for(int k=0; k<4; k++)
          {
               t.forward(100);
               t.turnLeft(90);
          }

          JFrame f = new JFrame();
          f.setSize(800,600);
          f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
          f.setContentPane(new Lab18());
          f.setVisible(true);
     }
     public void paintComponent(Graphics g)
     {
          g.drawImage(Turtle.getImage(),0,0,getWidth(),getHeight(),null);
     }
}
```



Specification
Open Unit2\Lab18\Turtle.java. Finish the Turtle class from the previous page.

Create Unit2\Lab18\Driver18.java. Use the driver shown above to draw a tilted square. Then modify the driver to draw a more interesting picture with your built-from-scratch turtles.

Sample Run
`Unit2Lab18.jar`

Exercises
Lab18

Write the methods `drawLine`, `drawCircle`, and `drawOval` using only the BufferedImage method `setRGB`.

```
public void drawLine(BufferedImage img, int x1, int y1, int x2, int y2)
{

}
```

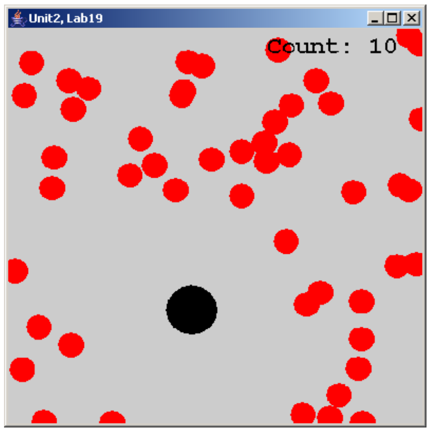
```
public void drawCircle(BufferedImage img, int x, int y, int r)
{

}
```

```
public void drawOval(BufferedImage img, int x, int y, int w, int h)
{

}
```

Lab19
Array of Polkadots

Objective
Multiple polkadots in arrays.

Background
Here is declaration of a Polkadot reference:



```
   private Polkadot pd;
```

Here is the instantiation of the Polkadot object using the 4-arg constructor:

```
   int x = (int)(Math.random() * N);
   int y = (int)(Math.random() * N);
   pd = new Polkadot(x, y, 25, Color.red);
```

Obviously, if we want 50 polkadots on the panel, we will want to use an array of polkadots. We need to declare the array reference:

```
      private Polkadot[] myPDarray;
```

And then create 50 objects in the panel's constructor with:

```
       myPDarray = new Polkadot[50];
      for(int k=0; k<myPDarray.length; k++)
         {
           x = (int)(Math.random() * N);
           y = (int)(Math.random() * N);
           myPDarray[k] = new Polkadot(x, y, 25, Color.RED);
         }
```

As you know, Polkadot objects know how to draw themselves. You will have to have each polkadot draw itself in the panel's Listener method.

The panel will also need a new `collide` method, something with the header

```
      public void collide(Ball b, Polkadot[] myPDarray)
```

This method will have to visit each polkadot and check to see if the distance between the ball and that polkadot is less than the sum of their respective radii.

Specification
Copy all files from Lab12. Modify PrizePanel to make an array of 50 (or 100, or 1000) polkadots. As before, when the ball hits a polkadot, the polkadot jumps to a new position.

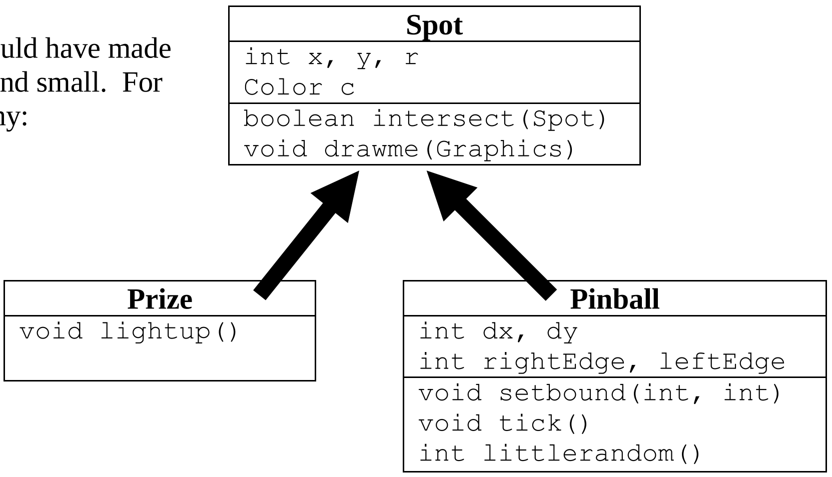
Create Unit2\Lab19\Driver19.java. Create an appropriate driver.

Sample Run
`Unit2Lab19.jar`

Lab20
Array of Prizes

Objective
Polkadots, balls, and arrays, all re-written.

Background
At every step of the way, the author could have made different design decisions, both large and small. For example, here is an alternative hierarchy:



In this version, prizes change color (from red to yellow) when they are hit by the pinball.

In this version, pinballs don't bounce at the same speed. Instead, they bounce in the reverse direction at a different speed--a new, small random number.

Here is the `actionPerformed` method of this version of PrizePanel:

```
private class Listener implements ActionListener
 {
  public void actionPerformed(ActionEvent e)
   {
     buffer.setColor(Color.WHITE);
     buffer.fillRect(0,0,N,N);
     for(int k=0; k<num; k++)
     {
       if(pb.intersect(array[k]))
          array[k].lightup();
       array[k].drawme(buffer);
     }
     pb.tick();
     pb.drawme(buffer);
     repaint();
   }
```

Specification
Write Spot.java, Prize.java, and Pinball.java to the specifications shown above.

Create Unit2\Lab20\PrizePanel.java. Write the panel. The Listener code is given above.

Create Unit2\Lab20\Lab20.java. Create the driver.

Sample Run
`Unit2Lab20.jar`