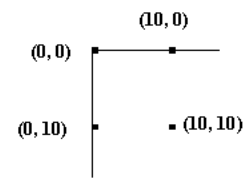
**Introduction to Graphics in Java**

Java is chock-full of classes that enable us to draw graphics on the screen. Some of these graphics come in the form of dialog windows such as those drawn by the methods showInputDialog and showMessageDialog. In later lectures, we will discuss how to create more elaborate graphical user interfaces, also known as GUIs (pronounced “goo’-eeez”).

Today we will begin by learning to create simple drawings of lines and circles, boxes and ovals, and color. It will involve making use of certain techniques that we have not yet learned about, but will also be explained in subsequent lectures. For example, you will encounter the term ‘*extends*’ which won’t mean anything to you now. Nor will a new datatype of object called ‘*Graphics*’. However, for now all you need to do is focus on which parts of the program can be modified to suit your needs, and accept that the other material can be learned/explained at a later point.

**Java’s Coordinate System**

To every GUI component on the screen such as a window, Java assigns the upper left hand corner a coordinate of (0, 0). As you move along the x-axis (to the right), the first number increases. As you move *down* (not up!) the y-axis, the second number increases. So the coordinate (20, 50) would be 20 pixels to the right and 50 pixels below the origin (0,0).



Coordinates are measured in pixels which is the unit of resolution used by your monitor.

**Drawing Lines, Rectangles, and Circles**

CODE: See the examples: FirstDrawing.java and TestFirstDrawing.java

A few initial notes:

* The class Graphics comes from the package java.awt which must, therefore, be imported.
* Classes JFrame and JPanel are both contained in the package javax.swing.
* The keyword ‘*extends*’ refers to a concept called ‘Inheritance’ which is beyond the scope of today’s lecture.

Most of the work we will do at this juncture will involve modifying the method paintComponent. In your driver class (the class that has the main method), the only things that will change are the line where you instantiate your test class: ClassName identifier = new ClassName(); and the method ‘setSize’ in which you must choose the size of the window you would like to draw on.

The method paintComponent

Every panel (essentially a graphics box) that you create has its own version of the method paintComponent. This method is *automatically* invoked for us by the Java system under different situations. For example, paintComponent is invoked:

* whenever the window is resized
* whenever the window is uncovered (if it was covered by a different window, or if it was minimized)
* the first time the program is run

The first statement in paintComponent method should always be:

super.paintComponent(g);

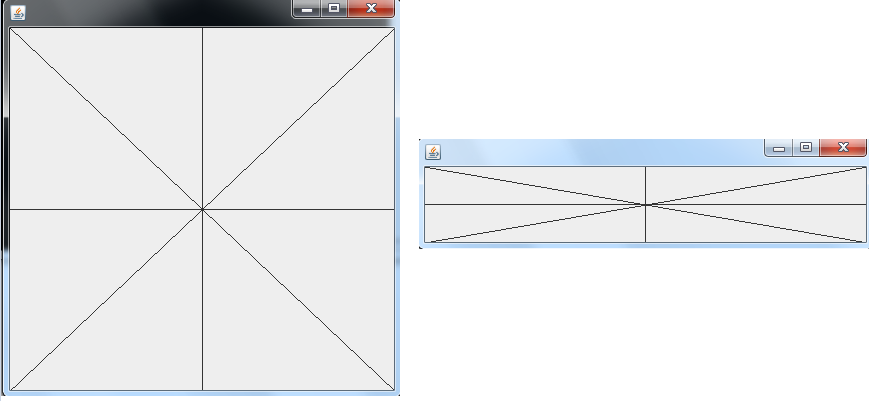
This is one of those times where we will have to defer explanation for later. Briefly, this command ensures that the panel is properly rendered before we begin drawing on it.

The Graphics class has several methods we can call in order to draw on our panel. Using the parameter ‘g’, we call the method drawLine. This method takes four arguments: The first two arguments represent the coordinates for the starting point of the line, and the last two arguments represent the coordinates of the end point of the line. For example, the method g.drawLine(0, 0, 50, 300) would draw a point from (0,0) – the upper left-hand corner of our panel, to the point (50,300) which is 50 pixels to the right and 300 pixels below the origin.

Note how we declare variables to keep track of the height and width of our panel. This makes it easier to envision what we are trying to accomplish when drawing our figures. We retrieve the width and height using the getWidth and getHeight methods. Though you will not find these methods in the current class, they do exist in a “parent class” which will be explained should you learn about inheritance. For now, just know that these methods return the width and height of the current panel.

Another important benefit to using the variables rather than the actual values 400 and 400 that we used in our driver class, is that if the user resizes our window, the lines will automatically readjust in size. (PS: Don’t forget that the paintComponent method is automatically invoked every time the window is resized).

Here is a drawing of the initial window (on the left) and the same window resized on the right. Notice how the lines are readjusted with all proportions remaining intact.



A quick explanation of the final few lines in our driver program:

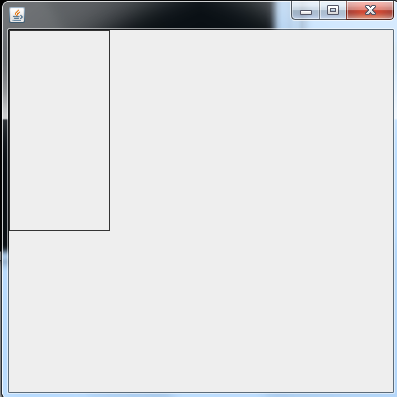
* The method setDefaultClose operation says what to do when the ‘X’ is clicked in the window. In this case, the command is to exit the program. There are other options which will be discussed at a later time.
* The ‘add’ method is required in order to add our panel to the window. Note that in Java parlance, what we typically think of as a window is actually called a ‘Frame’.
* Whenever you draw a frame (aka window!) you need to specify the size. Try commenting this line out and see what happens…
* Finally, the setVisible method is important because by default, frames are initially invisible.

Now let’s get a little bit creative and draw a series of fanning lines. This code is important in that it shows how we can use a simple loop and some very basic math to create a more interesting pattern. Note that coming up with this code does not happen immediately. Most people (myself included) would need to sit down with a pen and paper for a few minutes and draw things out along with the expected coordinates before trying to turn it into Java code.

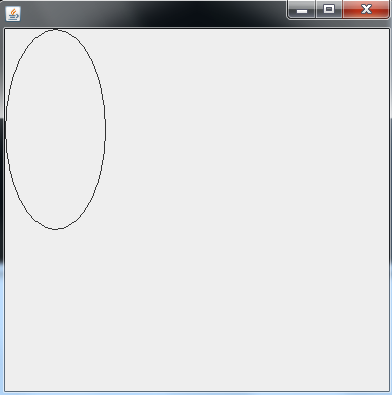
CODE: See the examples: FanningLines.java and FanningLinesDriver.java

The Graphics class also has methods to draw rectangles and ovals using the methods drawRect and drawOval respectively. There are also corresponding methods called fillRect and fillOval in which the shapes are not only drawn in outline, but are also filled in.

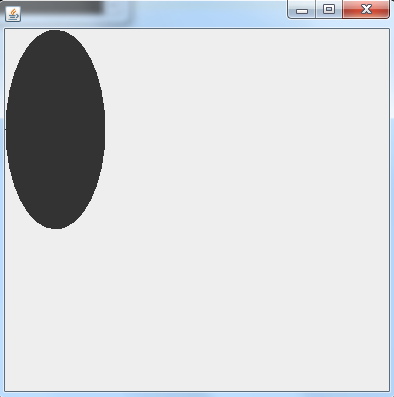
The rectangle methods require four arguments: The first two are the x and y coordinates of the upper left hand corner. The third argument is the width, and the fourth represents the height. So **drawRect(0,0, 100, 200)** will draw a rectangle beginning at the upper left-hand corner of the panel, extending 100 pixels to the right and 200 pixels down. An example is shown here:



The oval methods allow you to draw a circle or an oval. This method works by essentially asking you to provide coordinates for a rectangle. The program than draws a circle/oval such that the midpoint of each of the four ‘sides’ of the circle/oval touch the middle of the rectangle. Shown here is the result of drawing the same size rectangle as in the previous paragraph, but invoking drawOval as opposed to drawRect.



Here is a version with the same coordinates, but invoking fillOval:



There are many other lines that can be drawn including arcs by invoking drawArc and fillArc. Arcs are very convenient—imagine trying to draw a smile without one… Arcs are most easily understood if you simply think of them as a subsection of an oval. The fillArc and drawArc methods require six arguments.

The Java API for this method can be found at [this link](http://download.oracle.com/javase/6/docs/api/java/awt/Graphics.html#drawArc(int,%20int,%20int,%20int,%20int,%20int)) and is repeated here for your convenience:

**public void drawArc(int x, int y, int width, int height, int startAngle, int arcAngle)**

with the following parameters:

x - the *x* coordinate of the upper-left corner of the arc to be drawn.

y - the *y* coordinate of the upper-left corner of the arc to be drawn.

width - the width of the arc to be drawn.

height - the height of the arc to be drawn.

startAngle - the beginning angle.

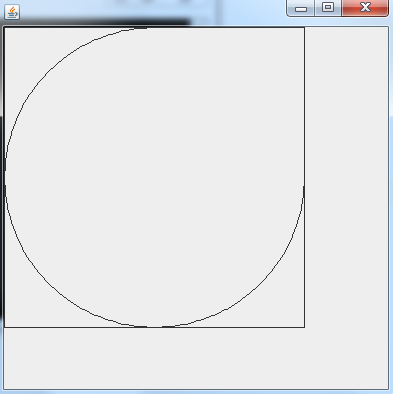
arcAngle - the angular extent of the arc, relative to the start angle.

So, the first four arguments represent the rectangle. The fifth and sixth are values in degrees. The fifth argument represents the starting angle on the oval and begins at the 3 o’clock position on the rectangle. The sixth is the “sweep” or amount of arc to cover.

The starting angle and sweep are in degrees. startAngle begins at the 3 o’clock position on the rectangle. A positive value for the sweep draws the arc in a clockwise direction. You can therefore provide a negative value to the sweep to draw the arc in a counterclockwise direction.

Examples: 360 would represent a full circle, 180 would represent a half-circle. -90 would represent a quarter-circle beginning at 3 o’clock and moving counterclockwise.

The example below shows a 300 by 300 rectangle drawn in a window that was originally 400x400. (The window has been shrunk to fit inside the current document). The arc goes counterclockwise from 0 to -270.



The command to generate this arc was: g.drawArc(0,0, 300,300,0, -270);

**Colors**

Not surprisingly, there is a great deal we can do with colors. Students of Java familiar with computer graphics will be happy to know that there are ways of using RGB values, Hexadecimal values, and other forms of color notation. Java also provides some extremely useful shortcuts to common and primary colors using constants in a class called, conveniently enough, *Color* which is contained in java.awt.

Using our Graphics object, we can invoke the method setColor and pass it any of many color constants including Color.RED, Color.CYAN, Color.BLUE, and several others such as ORANGE, YELLOW, GREEN, GRAY, LIGHT\_GRAY, etc. The full command to change the current color to gray would be: g.setColor(Color.LIGHT\_GRAY);

This will change all drawing to light gray until a new color is chosen.

**Layout**

When darwing graphics by ‘hand’, proper layout of items on your panel can be tricky until you get the hang of it. For this reason, I typically recommend drawing a rectangle around some of the items you are attempting to line up. This will help you align things properly, and make it easier to debug when things don’t line up the way you want them to.

CODE: See HappyFace.java and its corresponding driver