Graphics Operations in Java

Steven R. Vegdahl
CS 203 (Fall 2000, revised Spring 2001, Fall 2001, Spring 2008, Spring 2013, Fall 2015)

1.0 Introduction

Java provides a `Graphics` library that allows the programmer to draw shapes on a graphics window. These shapes are things like rectangles, ovals, polygons, and text. In addition, images (e.g., from a ".jpg" file) can be displayed.

Certain kinds of Java objects, such as `JPanel` (or `Canvas`) object, are graphical. Your program can draw something on them and, if the graphical object is visible, what you draw will show up on the screen. Every time such an object needs to display itself on the screen (e.g., because another window had been covering it, it invokes its `paint` method). You can define a behavior for the `paint` method by inserting a `paint` method into your graphical class, as in:

```
public class MyPanel extends JPanel {
    ...
   public void paint(Graphics g) {
    ...
   }
    ...
}
```

The definition of `paint` contains a `Graphics g` "parameter declaration" inside the parentheses; this defines `g` to be the name that `paint` uses to refer to object's graphics object. (We'll talk more about parameters later in the course. For now, just trust that you can use `g` inside the `paint` method.)

There are a number of `Graphics` operations that allow us to draw things on the `JPanel`. As an example, we can draw a rectangle with:

```
g.drawRect(20,30,60,100);
```

This will draw a rectangle whose top-left corner is at pixel location (20,30) on the graphical object, and is 60 pixels wide and 100 pixels high. (See below for a discussion of pixels.)

Important note:

> If you want to use Java's graphics operations, you should have the "import statement", `import java.awt.*;` near the top of your program, before your class definition(s). If you're defining a `JPanel`, you also need `import javax.swing.*;`.

2.0 Java's Graphical Coordinates

2.1 Pixels

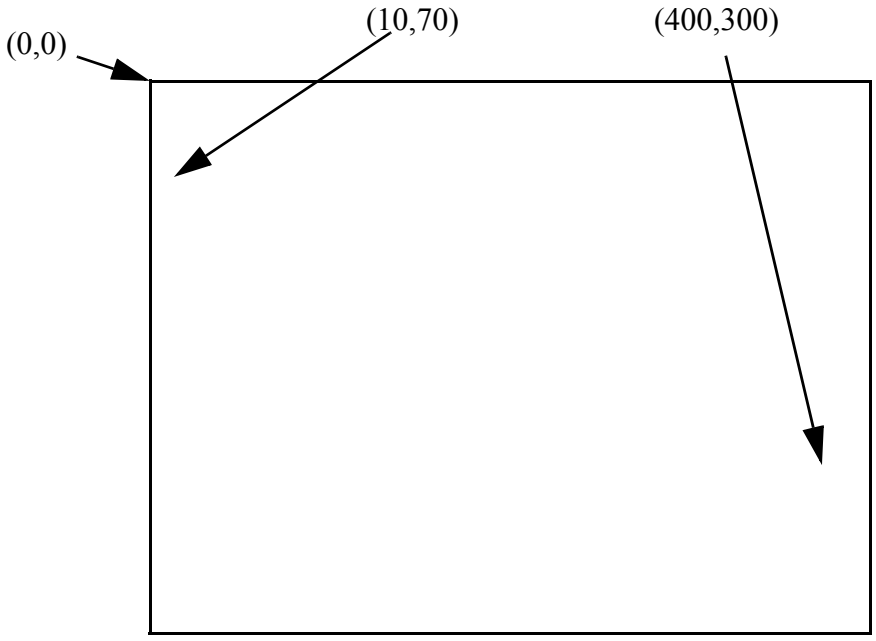
A Java `Graphics` object represents a rectangular region on the screen that consists of lots of itty-bitty dots called *pixels*. You might think of a screen as a "canvas" where any portion can be any color, but some of you—especially if you're a nerd—have gotten up close enough to the screen to realize that it the screen is actually a whole bunch of tiny dots. Each dot—call a *pixel*—can have its own color, but that's the finest resolution that the screen allows. If a screen has sufficiently high resolution, the fact that the pixels are there is generally not noticeable to the human eye. (However, you might notice it when thin lines are drawn that are neither vertical nor horizontal, and therefore appear "jaggy".)

People who sell monitors (i.e., computer screens) generally advertise the pixel-resolution. Typical numbers are 640x480 and 1280x1024.

2.2 Java's coordinate system

Java's coordinate system is defined in terms of pixels. For any graphics object, Java defines (0,0) to be the pixel at the top-left corner. Larger x-coordinate numbers indicate pixels farther to the right; larger y-coordinate numbers indicate pixels farther down on the screen. (Note that this is <u>not</u> the same as the Cartesian coordinate system in mathematics, where larger y-coordinate numbers mean going higher on the page.)

For example, the pixel at (1,0) would in the top row, one row to the right of the left edge of the window. The pixel at (400,300) would be the 400 pixels to the right and 300 pixels down from the top.



2.3 Pixel shape and layout

On many monitors, pixels are square, and are laid out on a square grid. On some, they are oval or rectangular, and may not be laid out on a square grid. If you have such a monitor, the program might think it's drawing a square because the pixel-height and pixel-width are the same; it would look like a rectangle on your monitor, however. In this discussion, we will assume that our pixels are laid out on a square grid.

2.4 Out-of-bounds pixels

A typical graphical window might be 300x400 pixels. You might ask what happens if drawing is done outside the region (e.g., at (500,700) or (-23,88)). The answer for Java is generally that anything drawn outside the visible region is not seen by the viewer of the window because it is not physically drawn. However, Java seems to use `short` integers—which "wrap around" at about 65,000—for coordinates. I have seen things show up in strange places when very large numbers are used.

3.0 Colors

A typical drawing contains more than one color. The Java library has predefined class—`Color`—that let us specify "all the colors of the rainbow" and a whole lot more.

3.1 Built-in colors

There are two basic ways to specify a color. First, the Java library has a handful of colors that are predefined, and may be referenced as "`Color`-dot-name":

- `Color.black`
- `Color.blue`
- `Color.cyan`
- `Color.darkGray`
- `Color.gray`
- `Color.green`
- `Color.lightGray`
- `Color.magenta`
- `Color.orange`
- `Color.pink`
- `Color.red`
- `Color.white`
- `Color.yellow`

Each of the above color-objects denotes the color suggested by its name.

3.2 Defining your own colors

Alternatively, new color objects can be created by specifying the amount of red, green and blue that they have in them. The numbers for each component should be in the range 0 through 255. If I wanted, I could create a color object representing a light purple by creating a color object with red and blue on fully, but no green at all:

```
Color lightPurple = new Color(255,0,255);
```

After this, the name `lightPurple` (*not* `Color.lightPurple`) could be used whenever I wanted the to specify the color light purple.

If you're not used to specifying a color using RGB (red, green, blue), it might be difficult for you to create the color you want. To aid you, there are web-sites that have lists of color-names and their corresponding RGB values. The following is a table adapted from a list produced by Douglas Adams at (the now-defunct) web page:

`http://www.sped.ukans.edu/~adams/norgb.htm`.

Table 1: Color Names and their RGB values (Douglas Adams)

color name	red value	green value	blue value
alice blue	240	248	255
antique white	250	235	215
aqua	0	255	255
aquamarine	127	255	212
azure	240	255	255
beige	245	245	220
bisque	255	228	196
black	0	0	0
blanched almond	255	235	205
blue	0	0	255
blue violet	138	43	226
brown	165	42	42
burly wood	222	184	135
cadet blue	95	158	160
chartreuse	127	255	0
chocolate	210	105	30
coral	255	127	80
cornflower blue	100	149	237
cornsilk	255	248	220
crimson	220	20	60
cyan	0	255	255
dark blue	0	0	139
dark cyan	0	139	139
dark goldenrod	184	134	11
dark gray	169	169	169
dark green	0	100	0
dark khaki	189	183	107
dark magenta	139	0	139
dark olive green	85	107	47
dark orange	255	140	0
dark orchid	153	50	204
dark red	139	0	0
dark salmon	233	150	122

Table 1: Color Names and their RGB values (Douglas Adams)

color name	red value	green value	blue value
dark sea green	143	188	143
dark slate blue	72	61	139
dark slate gray	47	79	79
dark turquoise	0	206	209
dark violet	148	0	211
deep pink	255	20	147
deep sky blue	0	191	255
dim gray	105	105	105
dodger blue	30	144	255
firebrick	178	34	34
floral white	255	250	240
forest green	34	139	34
fuchsia	255	0	255
gainsboro	220	220	220
ghost white	248	248	255
gold	255	215	0
goldenrod	218	165	32
gray	128	128	128
green	0	128	0
green yellow	173	255	47
honeydew	240	255	240
hot pink	255	105	180
indian red	205	92	92
indigo	75	0	130
ivory	255	255	240
khaki	240	230	140
lavender	230	230	250
lavender blush	255	240	245
lawn green	124	252	0
lemon chiffon	255	250	205
light blue	173	216	230
light coral	240	128	128
light cyan	224	255	255
light goldenrod yellow	250	250	210
light green	144	238	144
light grey	211	211	211
light pink	255	182	193
light salmon	255	160	122
light sea green	32	178	170
light sky blue	135	206	250
light slate gray	119	136	153
light steel blue	176	196	222
light yellow	255	255	224
lime	0	255	0
lime green	50	205	50
linen	250	240	230
magenta	255	0	255
maroon	128	0	0
medium aquamarine	102	205	170
medium blue	0	0	205
medium orchid	186	85	211

Table 1: Color Names and their RGB values (Douglas Adams)

color name	red value	green value	blue value
medium purple	147	112	219
medium sea green	60	179	113
medium slate blue	123	104	238
medium spring green	0	250	154
medium turquoise	72	209	204
medium violet red	199	21	133
midnight blue	25	25	112
mint cream	245	255	250
misty rose	255	228	225
moccasin	255	228	181
navajo white	255	222	173
navy	0	0	128
old lace	253	245	230
olive	128	128	0
olive drab	107	142	35
orange	255	165	0
orange red	255	69	0
orchid	218	112	214
pale goldenrod	238	232	170
pale green	152	251	152
pale turquoise	175	238	238
pale violet red	219	112	147
papaya whip	255	239	213
peach puff	255	218	185
peru	205	133	63
pink	255	192	203
plum	221	160	221
powder blue	176	224	230
purple	128	0	128
red	255	0	0
rosy brown	188	143	143
royal blue	65	105	225
saddle brown	139	69	19
salmon	250	128	114
sandy brown	244	164	96
sea green	46	139	87
seashell	255	245	238
sienna	160	82	45
silver	192	192	192
sky blue	135	206	235
slate blue	106	90	205
slate gray	112	128	144
snow	255	250	250
spring green	0	255	127
steel blue	70	130	180
tan	210	180	140
teal	0	128	128
thistle	216	191	216
tomato	255	99	71
turquoise	64	224	208
violet	238	130	238

Table 1: Color Names and their RGB values (Douglas Adams)

color name	red value	green value	blue value
wheat	245	222	179
white	255	255	255
white smoke	245	245	245
yellow	255	255	0
yellow green	154	205	50

Thus, if we wanted to create a `Color` object that denoted Douglas Adams' "sandy brown", we would make a Java declaration such as

```
Color sandyBrown = new Color(244,164,96);
```

3.3 The methods `darker` and `brighter`

There are two methods, darker and brighter, that can be applied to a color to create a new color object that is either darker or brighter (lighter) than the original. For example, a dark blue could be defined with:

```
Color darkBlue = Color.blue.darker();
```

Similarly, a very light green could be defined with

```
Color veryLightGreen = Color.green.brighter().brighter();
```

3.4 A note about color depth

Different monitor / video-card combinations will vary in the number of colors that they can actually show. (Sometimes, this is settable in software.) If you are running under a configuration with only 256 colors, you will not be able to exactly display most of the colors. Rather, you'll just get an approximation to each color. This means that if, for example, you want to display something that gradually goes from white to black, going through all the shades of gray, you might get only 10 different grays between the black and the white; your image will appear to be striped. If you are running with 24-bit color, you should not have this problem.

4.0 Java's Graphical Operations—Drawing Primitive Objects

Java has a number of operations that actually draw things on a window. Most operations take parameters that specify the pixel-location (and sometimes the size) of the object being displayed.

4.1 Specifying the color

A Java `Graphics` object has a *current color*, which the color it uses for drawing. If you don't like the color that the Graphics object is using, you can change it by invoking the `setColor` operation. For example the statement:

```
g.setColor(Color.blue);
```

tells the `Graphics` object "from now on—until you get another call to `setColor`—all of your drawing should be with the color blue". If you want to draw a "scene" that contains

objects with different colors, you will typically alternate between calls to `setColor` and the drawing operations that are described below.

Each `JPanel` object also has a *background color*. Anytime graphical object is repainted, its entire window is painted with the background color before any other drawing is done. When you're inside a method for an graphical class, the background color can be set using the `setBackground` method, as in:

```
setBackground(Color.gray); // not 'g.setBackground(...)'
```

Sometimes you may want to ask what the current color and background color are. You can do this with

```
Color myBackground = getBackground(); // not 'g.getBackground()'
Color myColor = g.getColor();
```

4.2 The `drawLine` operation

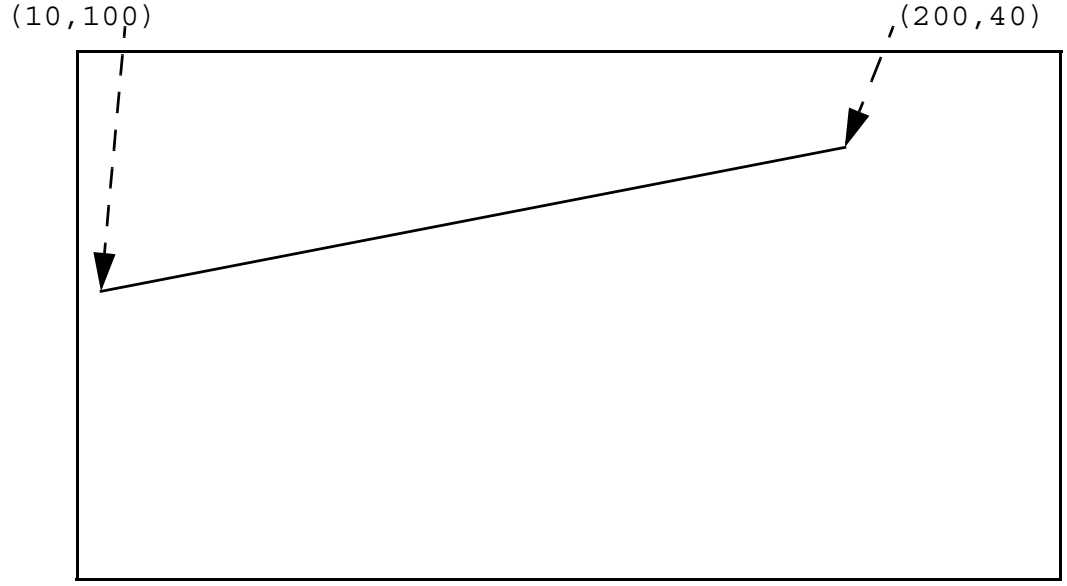
The `drawLine` operation draws a thin line (one pixel wide) on the window. It is invoked as follows:

```
g.drawLine(x1, y1, x2, y2);
```

where `x1`, `y1`, `x2` and `y2` are all integer values. It draws a straight line from the point `(x1,y1)` to the point `(x2,y2)`. For example, the operation

```
g.drawLine(10,100,200,40);
```

would draw a line from (10,100) to (200,40). This might look something like the following on the visible window:



4.3 The `drawRect` and `fillRect` operations

The `drawRect` and `fillRect` operations are used to draw rectangles. Both have the same parameters. The difference between them is that the `drawRect` draws a thin outline of a rectangle, while `fillRect` fills in the entire rectangular region.

A call to `drawRect` (or `fillRect`) is of the form:

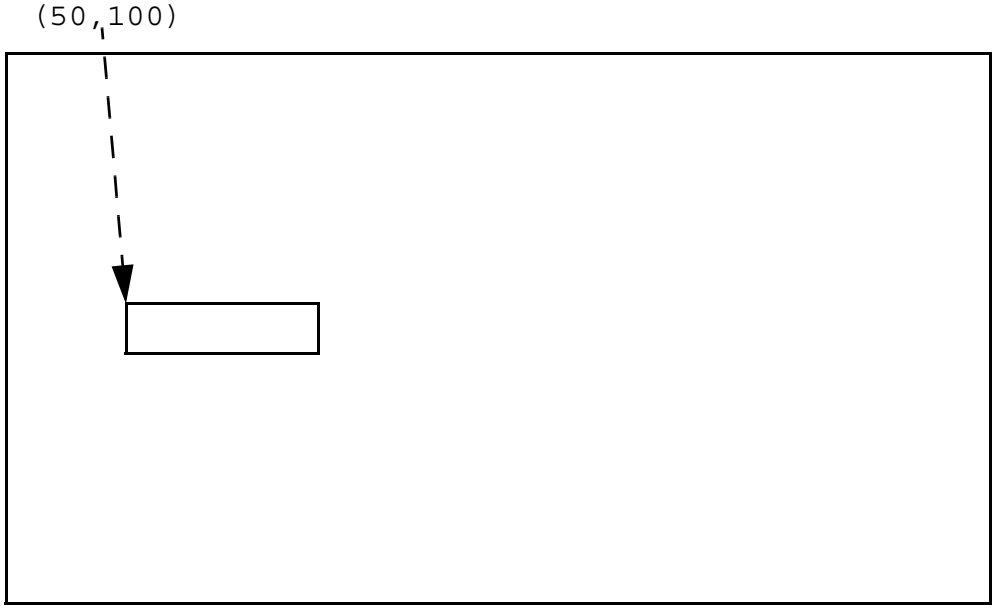
```
g.drawRect(x,y,w,h);
```

where (x,y) is the location of the top-left corner of the rectangle, w is the rectangle's width, and h is the rectangle's height.

Thus, the call

```
g.drawRect(50,100,80,20);
```

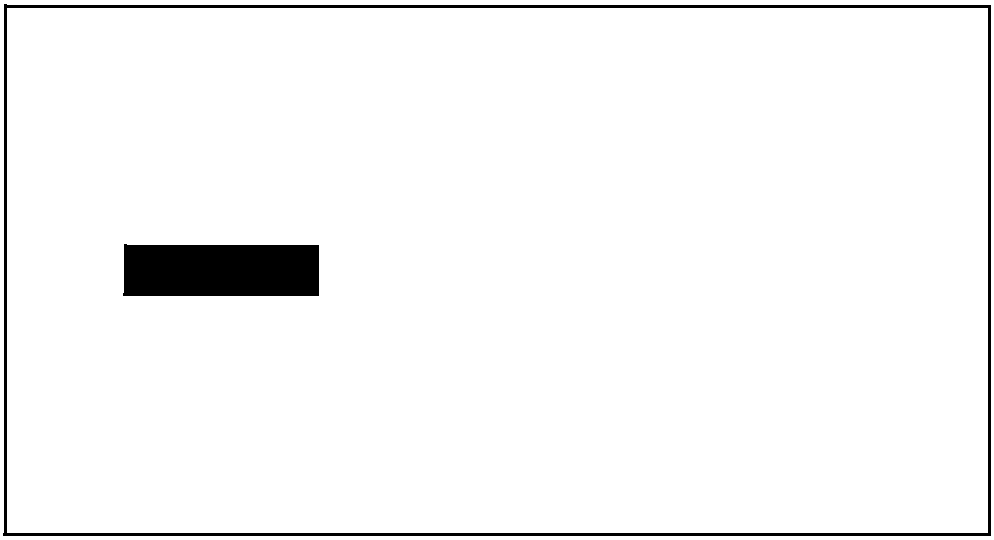
would cause visible window to look something like:



do be drawn. Similarly, the call

```
g.fillRect(50,100,80,20);
```

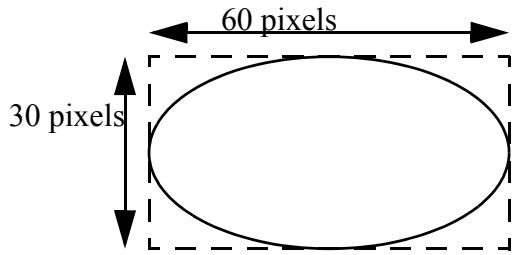
would cause the filled-in rectangle to be displayed, as in:



4.4 The `drawOval` and `fillOval` operations

The `drawOval` and `fillOval` operations are similar to `drawRect` and `fillRect`, except that rather than drawing the rectangle, they draw the oval that can be inscribed inside the corresponding rectangle.

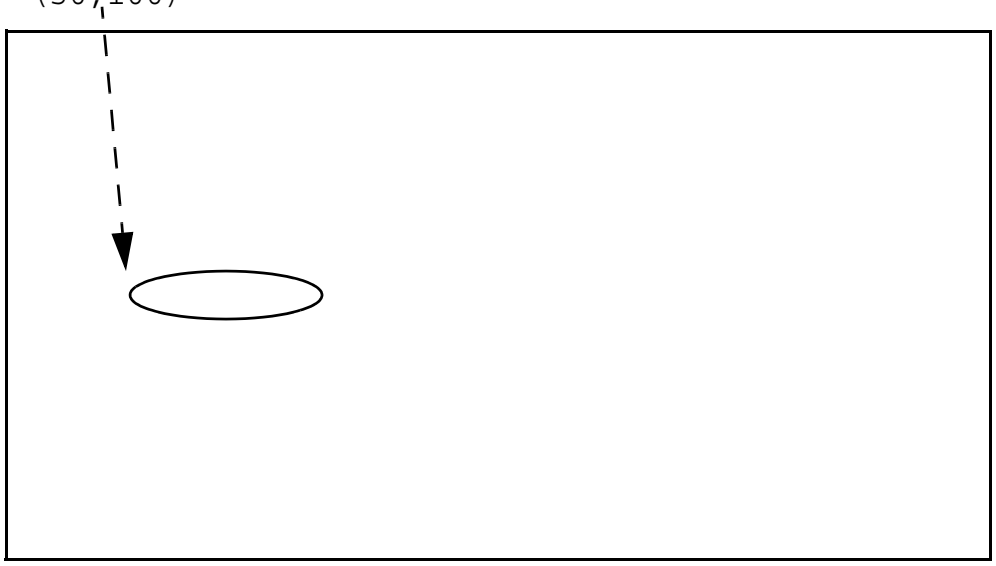
Thus, where `drawRect(...,...,60,30)` would draw a 30x60 rectangle, `drawOval(...,...,60,30)` would draw an oval whose width is 60 pixels at its widest point, and whose height is 30 pixels at its highest point, as in:



Thus, the call

```
g.drawOval(50,100,80,20);
```

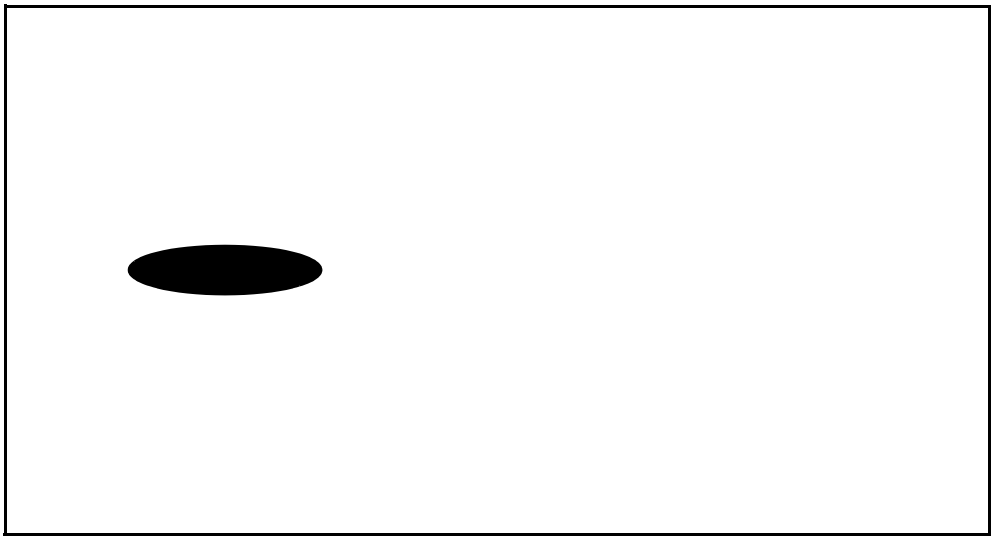
would cause visible window to look something like:



and the call

```
g.fillOval(50,100,80,20);
```

would display:



Note: there is no `drawCircle` (or `fillCircle`) operation. If you want to draw a circle, you just call `drawOval` (or `fillOval`), where the height and width values are identical.

When drawing small ovals, you will likely notice that they don't look terribly nice. This is due to the fact that we can physically draw only in full pixel increments. Thus, a 3x3 oval, when viewed closely, might look something like:

4.5 The `drawArc` and `fillArc` operations

The operations `drawArc` and `fillArc` are generalizations of `drawOval` and `fillOval`. Rather than drawing (or filling) the entire oval, they display only a portion of it. The portion that is drawn is specified by two extra parameters. In the call

```
g.drawArc(x,y,w,h,startAngle,numDegrees);
```

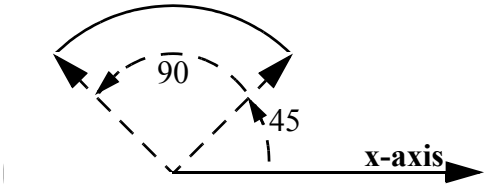
the first four parameters have the same meaning as in `drawOval`. The last two specify

- `startAngle`: the angle where the drawn portion of the oval starts. This uses standard polar coordinates (as in mathematics), so the angle 0 is "three o'clock", 90 is "12 o'clock, etc.
- `numDegrees`: the number of degrees of the oval that should be drawn. Here positive numbers indicate degrees in the counterclockwise direction; negative numbers indicate degrees in the clockwise direction (as in polar coordinates).

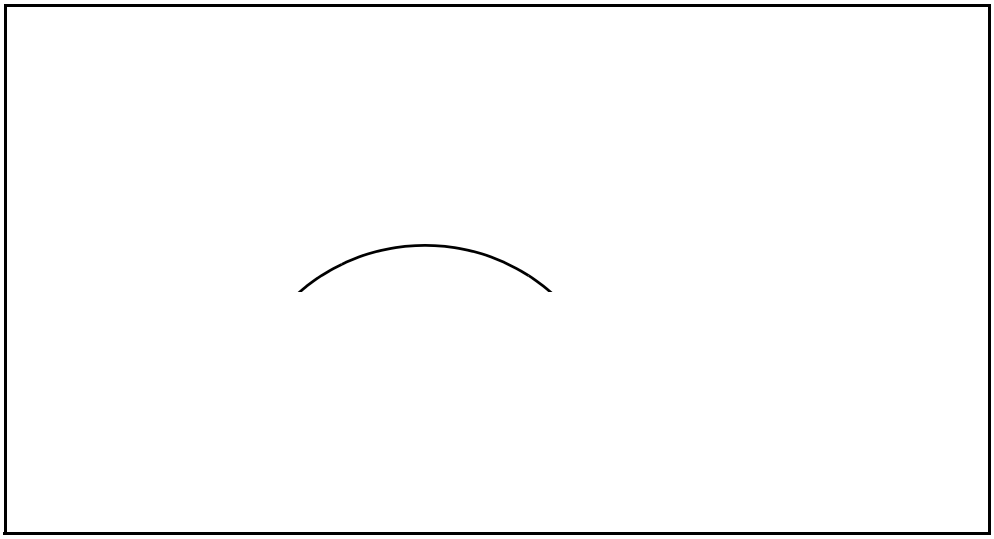
Thus, to draw the "top quarter" of a circle, we might issue the statement:

```
g.drawArc(100,100,200,200,45,90);
```

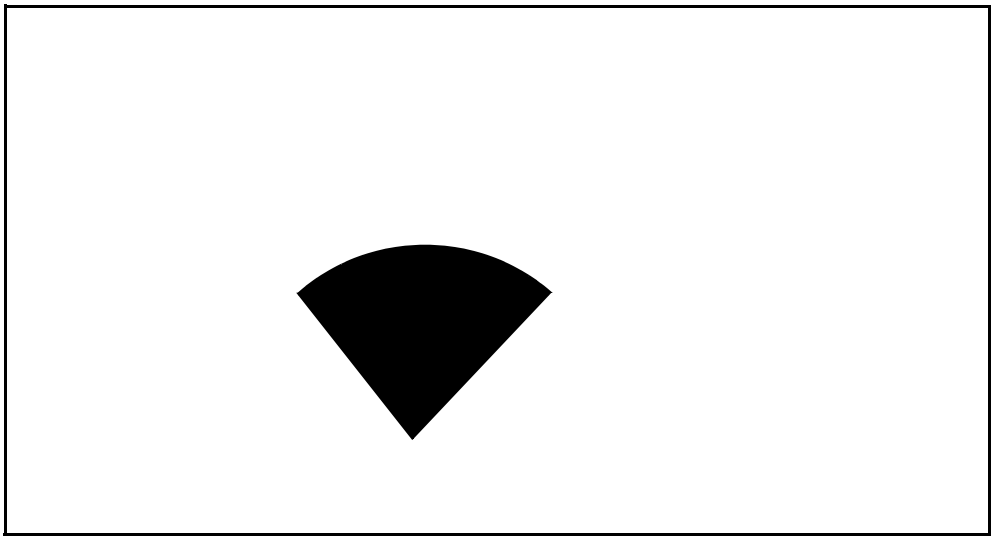
which would draw a portion of the circle that is inscribed in the 200x200 rectangle whose top-left corner is at (100,100). The portion that is drawn begins at the 45-degree mark, and extends counter-clockwise for 90 degrees, as in:



On the window, it would look something like:



The result of the corresponding `fillArc` operation is that of a "pie shape", as in:



4.6 Drawing and filling polygons

A polygon is a geometric object that consists of a number of flat sides. Special cases include triangles, parallelograms, pentagons and rectangles. Java's standard library defines a `Poly-`

gon class. You can create a `Polygon` object by specifying the points that define the endpoints of all its segments. This is done by creating an "empty Polygon" object, and then adding "points" to it, as in:

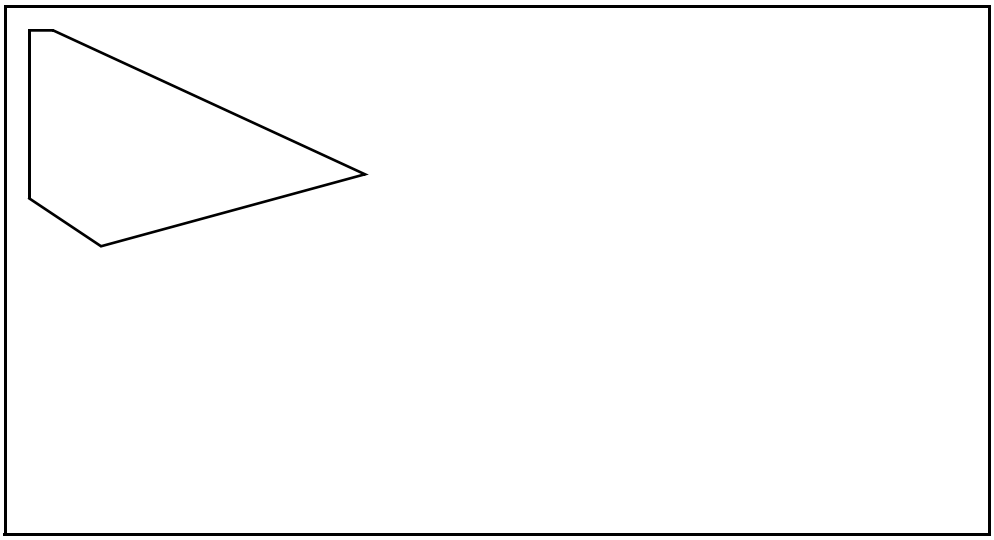
```
// create empty Polygon
Polygon myPolygon = new Polygon();

// add five points, to make a pentagon
myPolygon.addPoint(10,10);
myPolygon.addPoint(20,10);
myPolygon.addPoint(150,70);
myPolygon.addPoint(40,100);
myPolygon.addPoint(10,80);
```

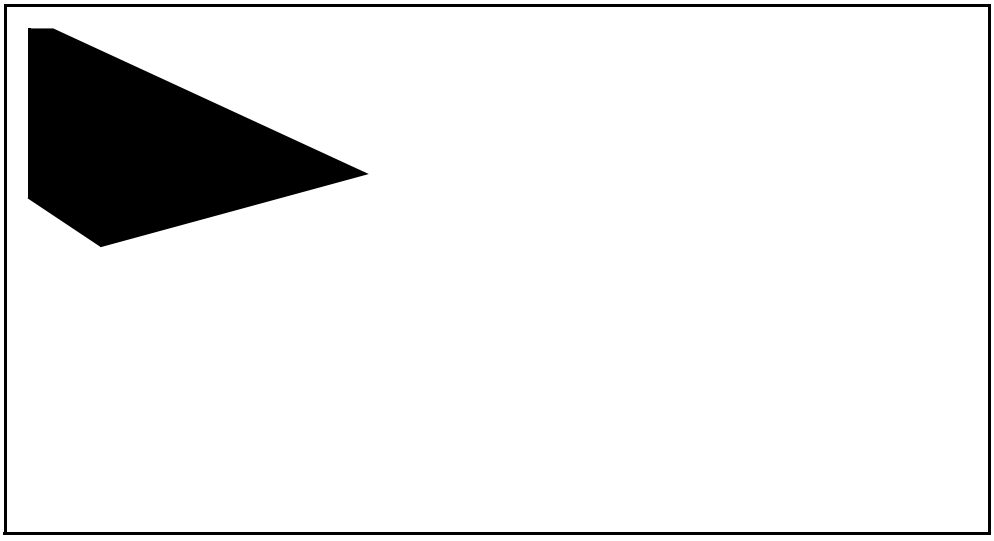
After such a polygon is created, we can draw (or fill) the `Polygon` onto a `Graphics` window, as in:

```
g.drawPolygon(myPolygon);
```

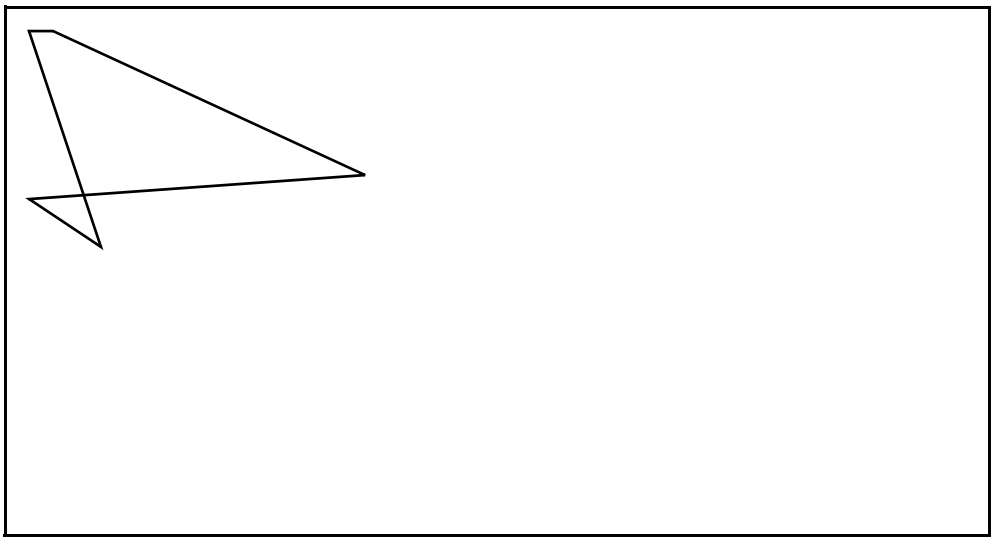
which would draw the following:



The corresponding call to `fillPolygon`, `g.fillPolygon(myPolygon);`, draws:



You must be careful to add the points in the order in which they occur as the `Polygon` is drawn. If you get them out of order, your Polygon's lines will start overlapping. For example, if the last two `addPoint` operations above had been reversed, the resulting `drawPolygon` operation would draw:



4.7 Drawing text on a graphics window

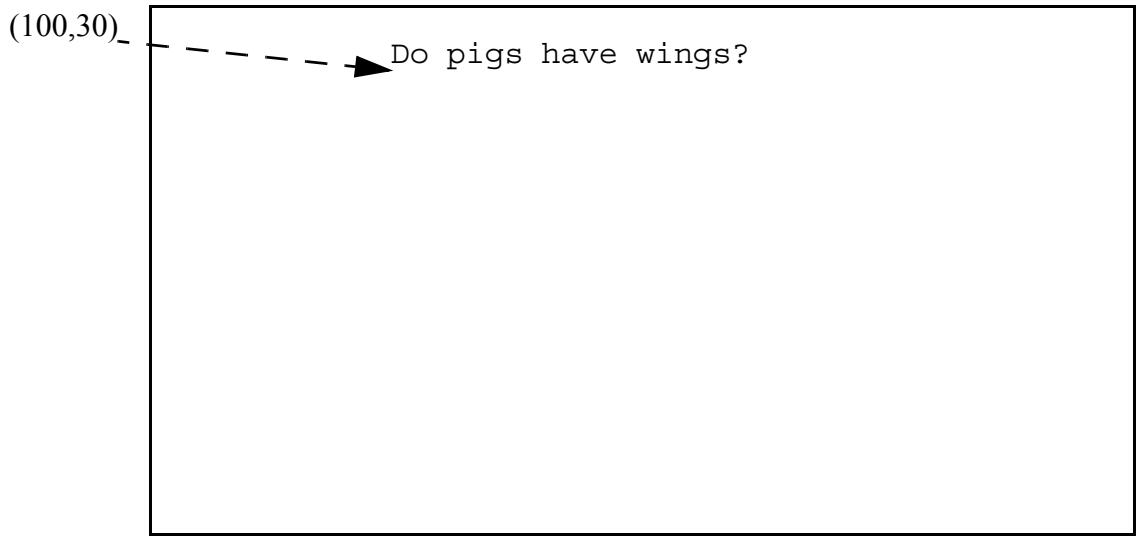
The `drawString` method is used to draw text onto a graphics window. Its three parameters are:

- the text `String` to be written onto the window.
- the `x` and `y` coordinates of the bottom-left corner of the (imaginary) rectangle in which the text is drawn.

As an example, the call:

```
g.drawString("Do pigs have wings?", 100, 30);
```

would a result in a window that looks something like:



You can change also change the font if you'd like. A `Graphics` object has a *current font*, which is the font it uses for all `drawString` operations. It can be changed using `setFont`.

Before you set the font, however, you need to create a font of the style and size that you want. You have three degrees of freedom in defining your font:

- a `String` that specifies its *font family*, which is one of
  - `"Serif"`, which specifies a font where most letters have serifs— "decorative lines" at edges and corners. For example, the 'T' shown here has its serifs circled:

  - `"SansSerif"`, which specifies a font where no letters have serifs. For example, the letter 'T' shown here has no serifs:

  - `"Monospaced"`, which is a font in which each character has the same width. This is useful if you want characters in a line of text to "line up" with corresponding characters in the line below it. (In the other two font families, different characters have different widths. For example, the letter 'i' is generally narrower than the letter 'M'.)
  - `"Dialog"`, which is presumably a font intended to be used in dialog boxes.
  - `"DialogInput"`, which is presumably a font intended to be used when a user types something into a dialog box.
- a *style*, which is one of `Font.PLAIN`, `Font.BOLD`, `Font.ITALIC`, or `Font.BOLD+Font.ITALIC`.
- its point-size, which is a positive number.

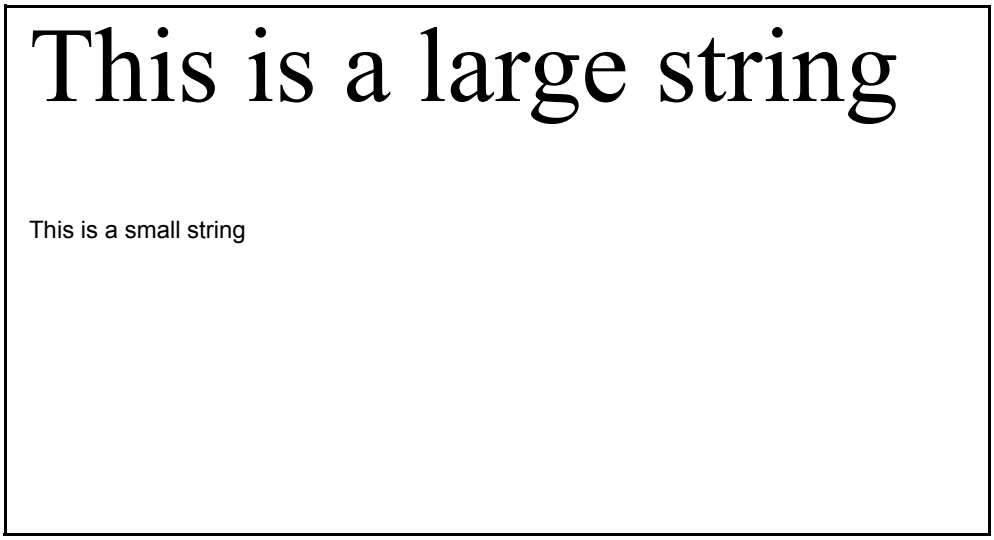
As an example, let's try writing two lines of text: a large `This is large text`, and a small `This is small text`. We can do this with:

```
// create our fonts
Font largeFont = new Font("Serif", Font.PLAIN, 40);
Font smallFont = new Font("SansSerif", Font.PLAIN, 9);

// draw the "large" string using the large font
g.setFont(largeFont);
g.drawString("This is a large string", 10, 50);

// draw the "small" string using the small font
g.setFont(smallFont);
g.drawString("This is a small string", 10, 100);
```

The resulting window would look something like:



5.0 Putting it All Together

In the following example, I have constructed a simple scene that contains a snowman labeled "Mr. Frosty". I've set the background to *sky blue* so that the white snowman can be seen against it. Mr. Frosty has two eyes of coal, a carrot nose, a black hat, and series of brown rocks for his mouth. It is done using a `Canvas` so that drawing can be seen during single-step debugging.

To do this, I had to figure out the coordinates on the screen where everything needs to be put—much of this was by trial and error. The comments in the program listed below will hopefully let you know what's going on.

(Note: the `translate` method, used below, moves a polygon from one place in the coordinate system to another without changing its shape.)

`Snowman.java`:

```
import java.awt.*;
import javax.swing.*;

public class Snowman extends Canvas {
    /**
     * init method, called when Snowman object is
     * first created: sets the background color and size
     */
    public void init() {
        // set background color to sky blue
        Color skyBlue = new Color(135,206,235);
        setBackground(skyBlue);

        // set the height and width of our graphical object so
        // that snowman is visible inside it
        setSize(400,400);
    }
```

```
/**
 * paint method: draws snowman
 *
 * @param g the graphics object
 */
public void paint(Graphics g) {

    // perform superclass painting
    super.paint(g);

    // draw the three circles that comprise Mr. Frosty's body
    g.setColor(Color.white); // the color of snow
    g.fillOval(80,200,200,160); // bottom ball, squished
    g.fillOval(115,110,140,140); // middle ball, round
    g.fillOval(140,50,80,80); // head, round

    // create an eye shape
    Polygon eye = new Polygon();
    eye.addPoint(8,0);
    eye.addPoint(14,8);
    eye.addPoint(12,12);
    eye.addPoint(2,11);
    eye.addPoint(0,4);

    // move eye to right-eye position; draw as black
    eye.translate(160,70); // moves polygon by (160,70)
    g.setColor(Color.black);
    g.fillPolygon(eye);

    // move eye to left-eye position; draw as black
    eye.translate(30,2); // moves polygon by (30,2) to (190,72)
    g.fillPolygon(eye);

    // draw carrot-nose (orange) as a small slice of an oval
    g.setColor(Color.orange);
    g.fillArc(175,60,100,100,150,20);

    // draw several small brown circular stones for the mouth
    Color brown = Color.orange.darker();
    g.setColor(brown);
    g.fillOval(165,108,5,5);
    g.fillOval(175,115,5,5);
    g.fillOval(185,118,5,5);
    g.fillOval(192,114,5,5);
    g.fillOval(200,108,5,5);

    // draw his black hat
    g.setColor(Color.black);
    g.fillRect(155,5,50,50); // base of hat
    g.fillRect(135,50,90,10); // rim of hat

    // give our picture a label
    g.setColor(Color.red); // use red for the label
    Font bigFont = new Font("SansSerif", Font.PLAIN, 25);
    g.setFont(bigFont);
    g.drawString("Mr. Frosty",250,50);
}
```

```
    /**
     * Snowman constructor, called when Snowman object is
     * first created: simply calls "init"
     */
    public Snowman() { // name must be same as name of class, above
        // call init()
        init();
    }

    /**
    * main method: creates frame and displays it
    *
    * @param args command line arguments
    */
   public static void main(String[] args) {
       JFrame f = new JFrame(); // create frame object
       Snowman s = new Snowman(); // create snowman object
       f.getContentPane().add(s); // put Snowman in frame
       f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE); // set up exit
       f.setSize(s.getWidth(), s.getHeight()+25); // set height/width
       f.setVisible(true); // display everything
   }
}
```

The resulting image in the window is: