An Introduction to Computer Science with Java

Last edited August 21, 2013 by C. Herbert

Introduction to Computer Science with Java

Chapter 11 – Java Graphics

This short chapter describes how programmers can create and manipulate two-dimensional graphics in Java. It includes creating.

Chapter Learning Outcomes

Upon completion of this chapter students should be able to:

- Describe what a *graphics processing unit (GPU)* is and why GPUs are used with modern computers.
- Describe what the *java.awt.Canvas* and *java.awt.Graphics* classes are and how they are used.
- Describe how to do each of the following using the Java Graphics class:
  - setting the Color for graphics;
  - drawing lines;
  - drawing rectangles, and rectangles with rounded corners;
  - drawing ovals, circles, and arc of ovals and circles;
  - drawing polygons;
  - drawing text on the screen.
- Describe how to create graphs and charts illustrating business and scientific data.
- Create software that uses the Java Graphics class to create figurative and abstract images.

11.1 Overview of Graphics Programming in Java

Graphics programming in Java is done through the use of APIs, including the *java.awt.Graphics* class and the Java 2D API that are both part of AWT; Java OpenGL (JOGL), Java 3D (J3D), and Java Advanced Imaging (JAI), which are official Java APIs; and many third party APIs, such as the Java LightWeight Game Library (JLWGL). The graphics capabilities in the AWT and the Java OpenGL graphics libraries are by far the most commonly used.

In this chapter we will focus on creating simple graphics and concepts of graphics programming using the graphics classes included in AWT, which will allow us to create and manipulate simple graphic images that can be displayed using the same containers as those used for Swing components.

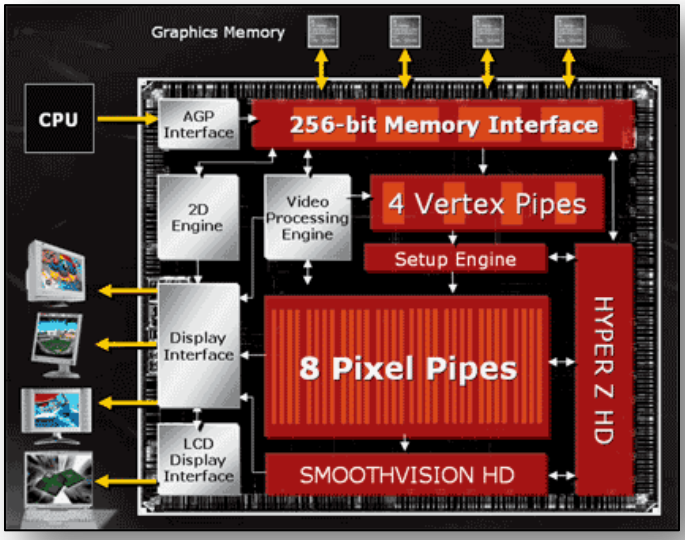
The graphics capabilities in AWT should work equally as well on Windows, Mac and Linux-based systems, although specialized coding for certain devices and unique hardware configurations may require the use of specialized APIs.

Generally, instructions in Java, as in any programming language are fed through the operating system to the computer's hardware. In the case of graphics programming, an additional component is often present – a graphics processing unit. A **graphics processing unit (GPU)**, often called a **graphics accelerator**, is a specialized processing unit for rendering high quality images and video on a computer screen. GPUs are often accompanied by additional specialized graphics memory on a board that can be added to a computer. Such boards are known as **graphics cards**.

GPU's are used because of the massive amount of data involved in imaging, especially video. A typical HD television image has between 900.000 and 2.1 million pixels per frame, with 3 or 4 bytes needed to store each pixel. At 30 frames per second, the highest quality raw video data would need a bandwidth in the neighborhood of 30 billion bytes per second. Many special techniques are used to reduce and compress this data. The Society of Motion Picture and Television Engineers (SMPTE) has developed the HD-SDI standard for broadcast HD video, which requires a bandwidth of 1.485 Gbits/Sec.

The image below shows a modern ATI Raedon HD GPU, which communicates with the CPU and 2 Gb of graphics memory, and can control several display monitors. Its components function as follows:

- the video processing engine does all of the computation in the GPU for 3D video graphics and can also perform 2D graphics.
- The 2D graphics unit is a simple graphics processor for 2D still images.
- the AGP interface communicates with the CPU.
- the memory interface communicates with the graphics memory.
- the vertex pipelines speeds communication between the memory and the video processing engine.
- the Hyper Z HD and Smoothvision HD units are proprietary circuits that manipulate images moving from video memory to the screen.
- The 8 pixel pipes are a series of data pipelines that translate graphic images to coordinate system for the hardware. They allow for parallel processing of data moving to display screens.



Even still images can require a lot of memory and processing power. A 3-megapixel image from a cellphone camera can use a megabyte of memory – which is as much as 8 minutes of MP3 quality sound or 64 thousand characters of Unicode text, roughly a 50 page document.

11.2 Using the *java.awt.Canvas* and *java.awt.Graphics* Classes

The most commonly used classes for graphics programming in Java are those found in AWT. They are often used in conjunction with Swing components in GUI programming. We will use two of those classes: the *java.awt.Canvas* class and the *java.awt.Graphics* class.

The **Graphics class** draws shapes on top of existing GUI components, such as drawing an image on a button. A **Canvas class** object in Java is a component representing a blank window on which we may draw shapes. We can also read user input on a Canvas, such as the location of the screen pointer when someone clicks a mouse button.

The *paint()* method is the method that draws on a component, but the *paint()* method in the Canvas class leaves the Canvas blank. We can create our own customized Canvas as a subclass of the Canvas class, then draw on our Canvas object by defining the *paint()*method for our class.

Inside the *paint()* method we will use methods from the *java.awt.Graphics* class to draw on our Canvas.

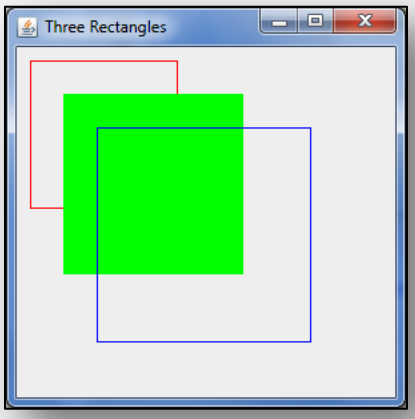
The *java.awt.Graphics* class contains many features for drawing images. Quoting from Oracle's Java Tutorials Website[1]: "*The Java 2D™ API is powerful and complex. However, the vast majority of uses for the Java 2D API utilize a small subset of its capabilities encapsulated in the java.awt.Graphics class".*

Here is an example of how this works. The following code creates a sub class of Canvas called *MyCanvas*, with three simple rectangles. The *paint()* method in the *MyCanvas* class will draw three rectangles on the canvas, the second of which is a filled rectangle. The objects are drawn in the order on which the code to do so appears in method, so the second rectangle is on top of the first, and the third in on top of the second. The *main()* method creates and displays a *JFrame* object containing a *MyCanvas* object.

```
/* ThreeRectangles.java
 * last edited Nov. 15, 2013 by C.Herbert
 *
 * This code demonstrates the creation of a canvas subclass for drawing
 * by overriding the paint method() in the subclass.
 *
 * The main() method displays the canvas with our graphihcs in a JFrame.
 *
 *
 */
package threerectangles;
import java.awt.*;
import javax.swing.*;

public class ThreeRectangles {

    public static void main(String[] args)
    {

        // create a MyCanvas object
```

[1] The java graphics tutorials are available online at: http://docs.oracle.com/javase/tutorial/2d/basic2d/index.html They are useful, but may be difficult to use for beginning programmers.

```
        MyCanvas canvas1 = new MyCanvas();

        // set up a JFrame tpo hold the canvas
        JFrame frame = new JFrame();
        frame.setSize(400, 400);
        frame.setLocation(200, 200);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        // add the canvas to the frame as a content panel
        frame.getContentPane().add(canvas1);
        frame.setVisible(true);

    } // end main()

} // end class ThreeRectangles


class MyCanvas extends Canvas
{
    public MyCanvas()
    {
    }  // end MyCanvas() constructor

    public void paint(Graphics graphics)
    // note: the graphics parameter is required as an artifact from inheritance
    {
        graphics.setColor(Color.red);
        graphics.drawRect(10, 10, 110, 110);

        graphics.setColor(Color.green);
        graphics.fillRect(35, 35, 135, 135);

        graphics.setColor(Color.blue);
        graphics.drawRect(60, 60, 160, 160);
    }  // end paint()

} // end class MyCanvas
```



In summary, methods from the Graphics class (*java.awt.Graphics*) such as the *drawRect()* method, can be used to draw on a GUI component. The Canvas class (*java.awt.Canvas*) is a component that creates a blank canvas just for such drawing. The paint *method()* in the Canvas class leaves the canvas blank. We can create our own canvas as a sub-class of the Canvas class and draw on the canvas by overriding the *paint()* method. The canvas can then be displayed by placing in a container, such as a JFrame object, and making the container visible.

11.3 Using Methods from the Graphics Class

In this section we will see how to draw with some of the most commonly used methods from the Graphics class, including methods for:

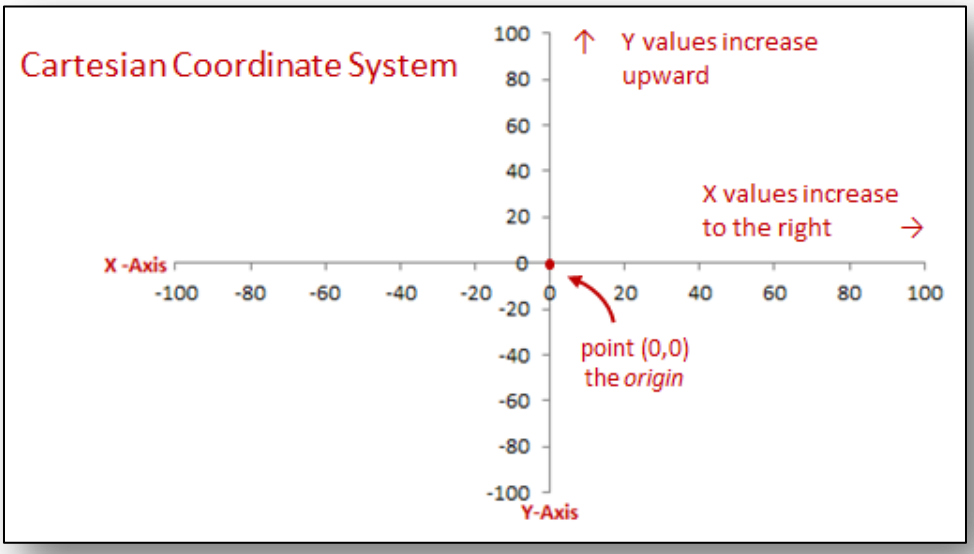
- setting the Color for graphics;

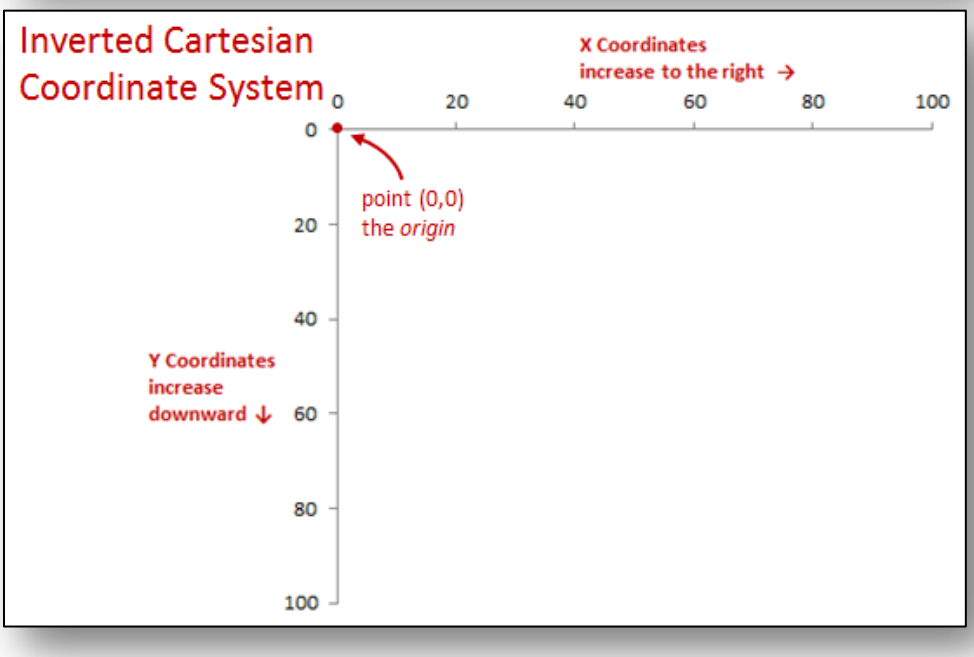
- drawing lines;
- drawing rectangles, and rectangles with rounded corners;
- drawing ovals, circles, and arc of ovals and circles;
- drawing polygons;
- drawing text on the screen.

We will start by looking at the coordinate system for onscreen graphics.

Onscreen Coordinates

Most computer screens use an inverted Cartesian coordinate system A Cartesian coordinate system – named for its inventor René Descartes – is a standard two-dimensional rectangular coordinate system with an x values that increase to the right and y values that increase upward. The two-dimensional system is bisected horizontally by a vertical y-axis where x =0 and a horizontal x-axis where y=0, as shown below. Points are located with an ordered pair of x and y coordinates – (x,y).





An inverted Cartesian coordinate system is the same as a Cartesian coordinate system, except that the Y-axis values increase with movement downward. The first computer screens that appeared in the middle of the Twentieth Century displayed text in lines across the screen with one line below another starting from the top left corner of the screen. The text coordinates were measured from the top left corner across and down.

Graphics on computer monitors followed this convention, with the origin in the top left corner of the screen, the x-coordinate increasing across the screen from left to right, and the y-coordinate increasing down the screen from top to bottom. Thus the name *inverted Cartesian coordinate system*, in which the y coordinate goes up as we move down the screen.

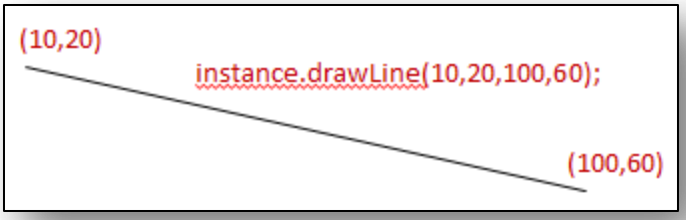
In containers such as *JFrames*, the same *inverted Cartesian coordinate system* is used. Points are referenced by (x,y) ordered pairs, with the origin in the top right corner of the frame. The x coordinate increase to the right and the y coordinate increase downward.

The following sections describe some of the most commonly used methods from the Graphics class. The methods would be used in the paint method to draw on a component, such as a class that extends the Canvas class, as in the example on pages 4 and 5 above.

In these examples, *instance* is the name of the instance of the graphics class. All coordinates are integer values of data type *int.*

Drawing Lines

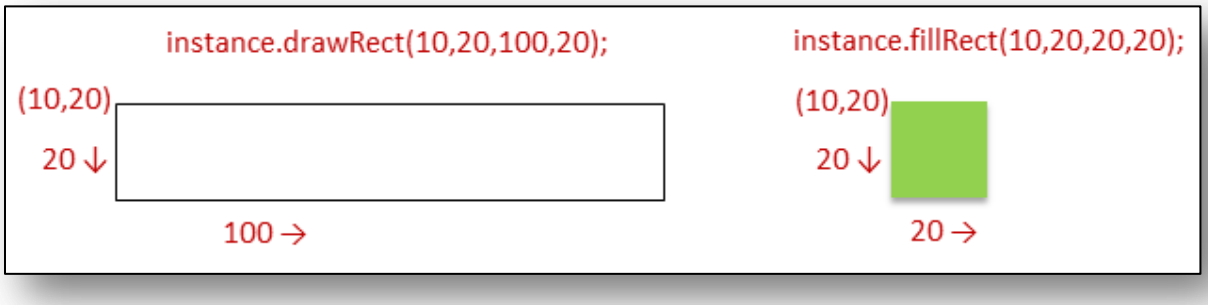
drawLine(int x1, int y1, int x2, int y2);
draws a line between the points (x1, y1) and (x2, y2).



Drawing Rectangles

drawRect(int x, int y, int width, int height) draws the outline of a rectangle of the specified width and height with the top left corner at point (x,y). The bottom right corner will be at point(x+width, y+height. To draw a square, specify equal values for width and height.

fillRect(int x, int y, int width, int height) draws a filled rectangle.

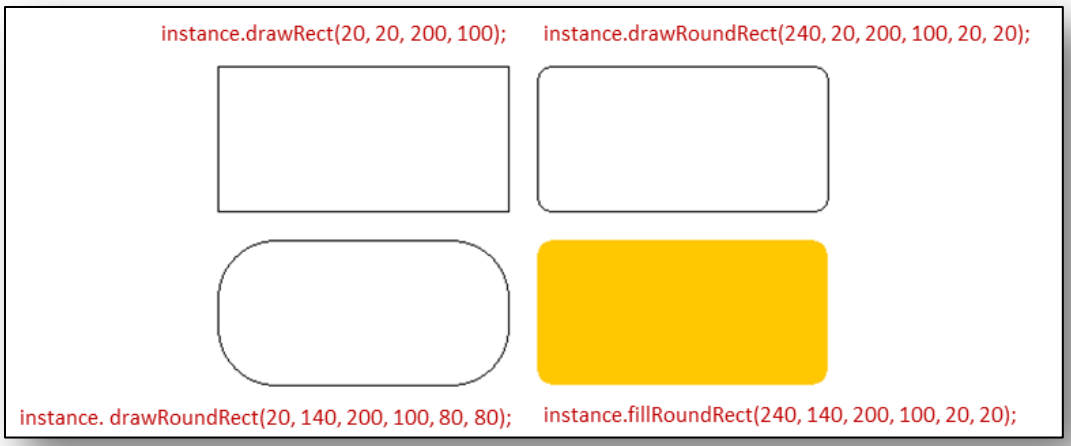


Rectangles with Rounded Corners

drawRoundRect(int x, int y, int width, int height, int arcWidth, int arcHeight) draws an outline of rectangle with rounded-corners. The *arcWidth* and *arcHeight* parameters set the horizontal and vertical diameter of the arc used for the corners of the rectangle.

fillRoundRect(int x, int y, int width, int height, int arcWidth, int arcHeight) draws a filled rectangle.

The example below shows a standard rectangle and three rectangles with rounded corners, each drawn with different parameters.



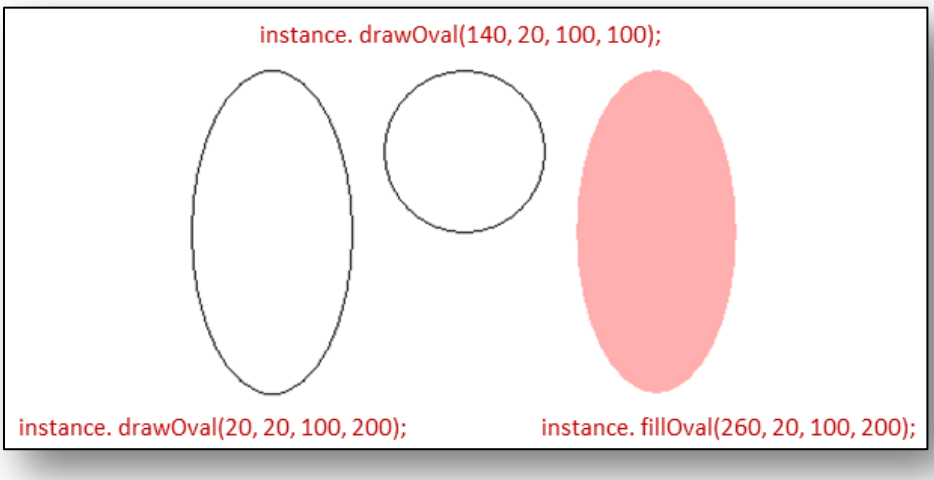
Drawing Ovals and Circles

drawOval(int x, int y, int width, int height) draws an outline of an oval within a rectangular area whose top left corner is at point (x, y) and whose bottom right corner is at point (x+width, y+height).

To draw a circle, specify equal values for width and height. The diameter of the circle will be equal to the value used for width and height; the radius will be half that value. The center point of an oval or a circle will be at coordinates (x+(width/2), y+(height/2)).

fillOval(int x, int y, int width, int height) draws a filled oval.

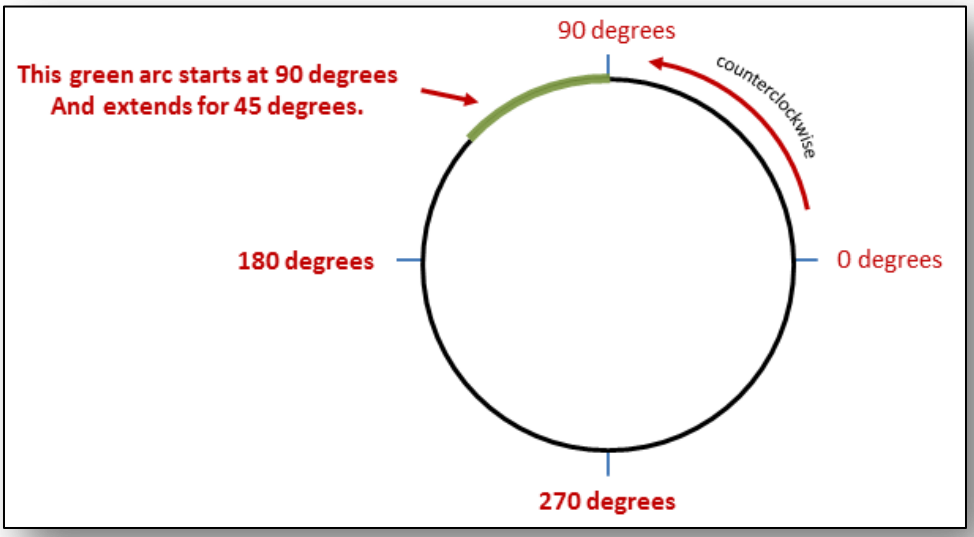
The example below shows three ovals, each drawn with different parameters.



Drawing Arcs of Ovals and Circles

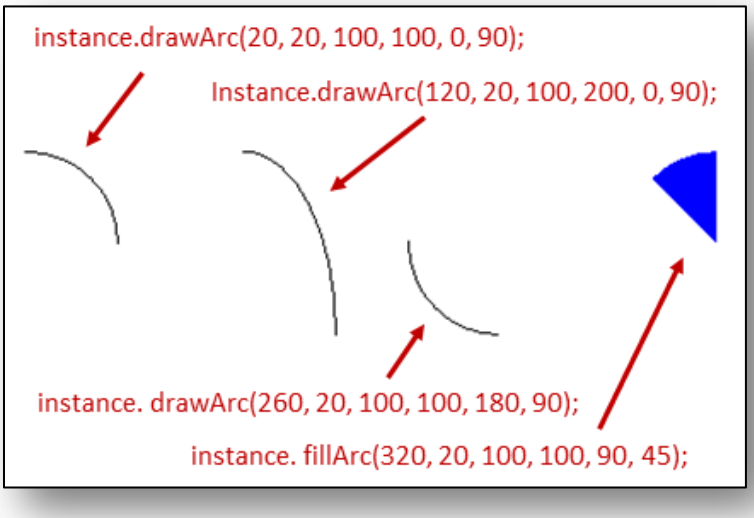
drawArc(int x, int y, int width, int height, int startAngle, int arcAngle) draws an outline of arc, which is a segment of the perimeter of an oval where:

- the complete oval is described within a rectangular area whose top left corner is at point (x, y) and whose bottom right corner is at point (x+width, y+height);
- the arc begins at *startAngle* degrees along the oval, and extends for *arcAngle* degrees;
- the oval is oriented so the 0 degrees faces to the right (east), and measurement moves counterclockwise around the oval for 360 degrees, as shown in the diagram below.



fillArc(int x, int y, int width, int height, int startAngle, int arcAngle) draws a filled segment of an oval defined by the specified arc.

The code below draws the arcs shown, each drawn with different parameters.



Drawing Polygons

drawPolygon(int[] xPoints, int[] yPoints, int n) draws a polygon with *n* vertex points defined by an array of *x* coordinates, and an array *y* coordinates.

fillPolygon(int[] xPoints, int[] yPoints, int nPoints) draws a filled polygon.

draw a polygon with 4 vertex points
(20,20) (100,20) (140,80) (60,80)

```
int n = 4;
int[] x = {20, 100, 140, 60};
int[] y = {20, 20, 80, 80};
instance.drawPolygon(x, y, n);
```

draw a polygon with 6 vertex points
(30, 40) (40,20) (60,20) (70,40) (60, 60) (40,60)

```
int n = 6;
int[] x = {30, 40, 60, 70, 60, 40};
int[] y = {40, 20, 20, 40, 60, 60};
instance.fillPolygon(x, y, n);
```

Drawing Text

drawString(String str, int x, int y) draws text specified by given String, starting with the baseline of the leftmost character at point (x,y).

The Color and Font of the text can be specified using the *setColor()*and *SetFont()* methods for components, as discussed in chapter 7, section 7.9.

The example below shows code to draw a text message identifying a previously drawn shape.

```
instance.setColor( new Color(153, 153, 255) );
instance.setFont(n ew Font("Cambria", Font.BOLD, 18) );
instance.drawString("This is a hexagon", 80, 80 );
```

This is a hexagon

Drawing Images from a Data File

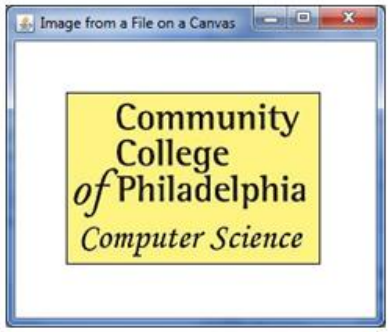
drawImage(Image img, int x, int y, ImageObserver observer) draws the image *img* with its top left corner at the point (x,y). *observer* can be left null. Two lines are required to create an image from a file and display the image:

```
Image name = new ImageIcon( "filename" ).getImage();
instance.drawImage(name, x, y,null);
```

The first line will create an image with the name *name* from the file identified by the String *"filename"*, where *"filename"* may be a local or full context name of an image file. The second line will draw the image on the screen. This technique works with *JPEG*, *PNG*, *GIF*, *BMP* and *WBMP* file types, although there may be some timing issues loading animated GIF files as Java Image class objects.

The following shows an example of this:

```
Image logo = new ImageIcon("CSCI.jpg").getImage();
graphics.drawImage(logo, 40, 40,null);
```



A sample Netbeans project named *DrawDemo* that demonstrates the commands in this section is included in the files for this chapter.

11.4 Programming Examples – a Histogram

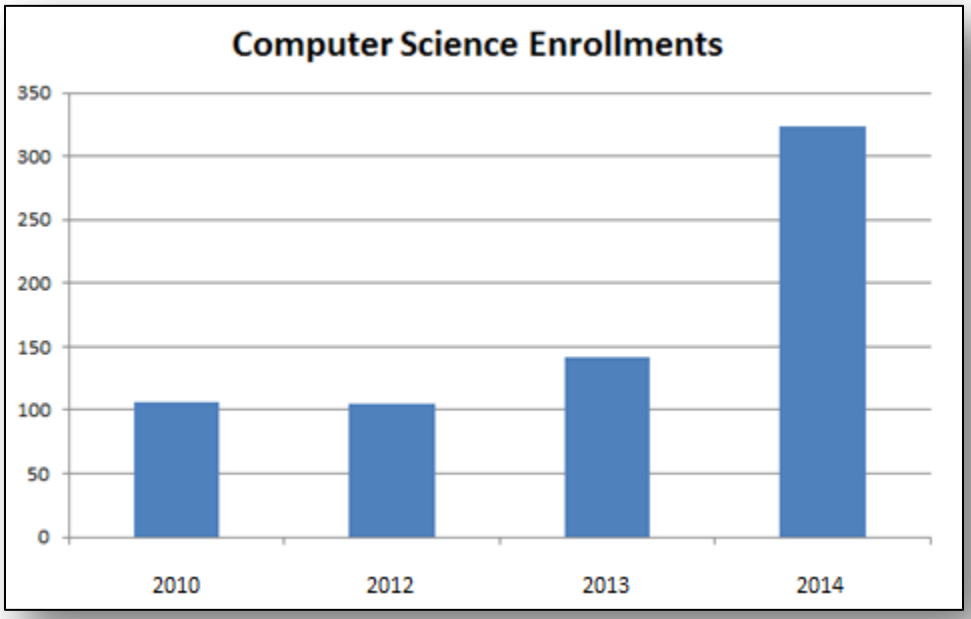
A **histogram** is a graphic representation of data, also known as a bar chart. In this example we will look at a histogram created using java's graphics class.

The first step in drawing any graphic image is to design the image. We wish to create a bar chart showing enrollments in Computer Science courses at Community College of Philadelphia in the Fall semester for four years. Here is the data:

year	students
2010	106
2011	105
2012	142
2013	324

Sections were added in 2012 and again in 2013, increasing enrollment.

We wish to create a histogram that looks something like this one, created in Microsoft Excel:



We need to create text labels, rectangles to represent the data, and the lines on the chart. The size of each rectangle is related to the magnitude of the data. The maximum data value is 324. We will use a slightly higher number, 350, for the scale on our graph – 0 to 350. Each of our bars will be proportional in size to 350.

year	students	
2010	106	106/350 = .303
2011	105	105/350 = .300
2012	142	142/350 = .406
2013	324	324/350 = .926

Our scale for the graph will be 20 pixels for each 50 units of data. This means that 350 units on the y-axis will take up 140 pixels. They will be drawn up from the bottom, with a baseline of y = 200.

The bars will be 20 pixels wide, with 20 pixels between bars.

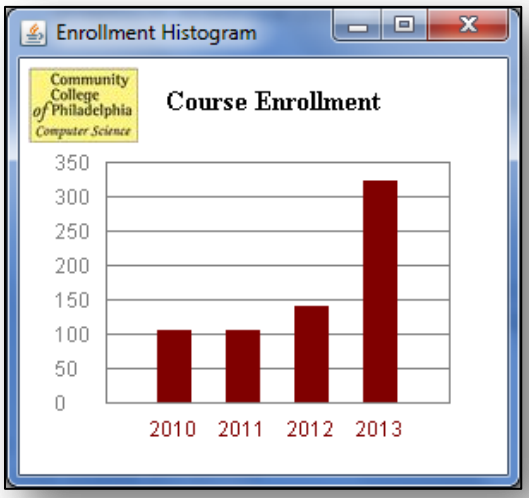
We will also put a small picture of the Computer Science logo in the upper left corner of the window.

The program needs to do the following:

1. place logo in corner
2. place title at top of chart
3. in a loop, place the horizontal lines and the labels for the lines 50, 100, 150, etc. up to 350.
4. in a loop, draw the bars – every 40 units in the x direction, 20 units wide, with the height of the filled rectangle for the bar determined by the enrollment value for that bar.

The program has some math to calculate where things should be drawn on the canvas, and to adjust how the labels line up with lines and bars on the chart.

Here is what the finished histogram looks like:



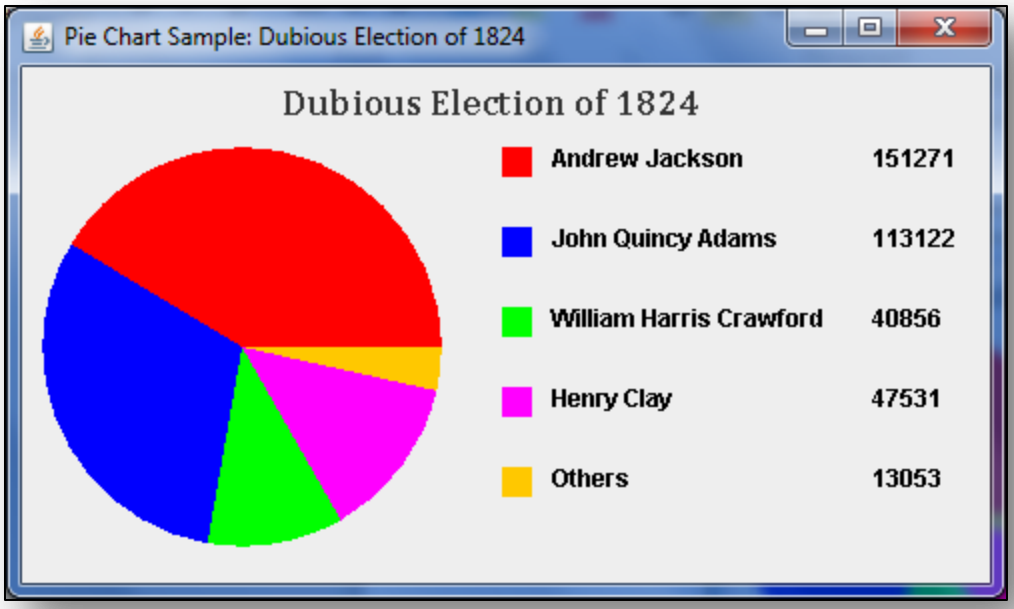
The lines, labels and filled rectangles for the bars are each drawn using standard Graphics class statements. The difficult part is laying the design of the chart, then doing a bit of tedious math to figure out the coordinates for drawing the objects. The inverted Cartesian screen coordinates make things a bit more tedious. Using loops to draw the lines and the bars saves some work.

The code to draw this histogram is included in the NetBeans project *Histogram*, included with the files for this chapter. Obviously, it is easier to use software such as Excel to draw charts and graphs. This project is included as an object of study to learn more about using Graphics class objects in Java.

11.5 Programming Example – a Pie Chart

A pie chart, like the one shown below, depicts how each number in a set of numbers is part of the whole in proportion to the other parts. Pie charts can be created in Java by using the *fillArc()*method. Adjacent filled arcs are drawn for each value with the length of the arc proportional to the size of the value for that arc.

For a pie chart, we need to know where each arc starts and the size of the arc in degrees. The total pie chart will be 360 degrees. First we calculate the sum of the values. Then, The size of the arc (in degrees) that corresponds to each value will be *(value* 360/sum)*. The first arc will start at zero degrees. For each slice of the pie chart, we will calculate the size of the arc, draw the arc, then use the size to calculate the starting angle for the next arc.



Each slice of the pie chart will have a value, a label for that value, and a color. The label and value can be read in from a data file or entered by a user. The colors can be defined in the program.

We will also need a title for our pie chart. In the example below, we will read data from a file that has: the title of the chart on the first line, then sets of lines – each having a line with a label, followed by a line with the value that matches the label.

In addition to drawing the pie chart, we will draw a legend that has a square with the same color as the slice, along with the label and value for that color.

The following algorithm shows how to do this:

1. read the data file:
    a. first, read the title
    b. then in a loop,
    c. Load the labels and values for each slice into arrays. We will put the labels in the array *sliceLabel[].* We will put the values in the array *sliceValue[].* Alternatively, this could be done by letting the user enter the data.
    d. As we load the values, we will also calculate the sum of the values.
2. We will also need a color for each slice. We will call the array *sliceColor[].*We can hardcode these colors into our program. *Color[] sliceColor = { Color.RED, Color.BLUE, Color.GREEN, Color.MAGENTA, Color. PINK, Color. GRAY, Color. CYAN, Color. ORANGE}* This array of colors can handle up to 8 slices in the pie chart. Alternatively, we could define our own Color for any slice, by using a Color constructor with RGB values, such as *new Color(128,0, 255)*.
3. Draw the Title for the chart.
4. Draw the body of the chart. We need to keep track of where each arc starts and the size of the arc in degrees. We will initialize the starting angle to be 0 degrees. The total pie chart will be

360 degrees. The size of the arc (in degrees) corresponding to each value will be *(sliceValue [i] * 360/sum)*. In a for loop for the number of slices:

a. Calculate the size of the arc: (size = *(a[i] * 360/sum) * 360)*
b. Set the color for arc i to be *sliceColor[i].*
c. Draw the arc: *fillArc(x, y, width, height, start, size)*
d. Draw the square in the legend for this slice.
e. Add the label and value next to the square.
f. Calculate the value of start for the next slice: start = start + size.

When the loop is finished, the chart will be done.

The code for this is included as the NetBeans project *PieChart* in the files for this chapter. It draws the pie chart above, showing the popular vote for the *Dubious Presidential Election of 1824*, which resulted in the House of Representatives selecting the president in what has come to be known as the "*Corrupt Bargain*". You can draw pie charts for different data simply by editing the *PieChartData.txt* file included with the project.

Data about each slice of the pie can be stored in parallel arrays, or as properties of a *Slice* object. The example for this uses parallel arrays.

11.6 Programming Example – Trigonometric Functions

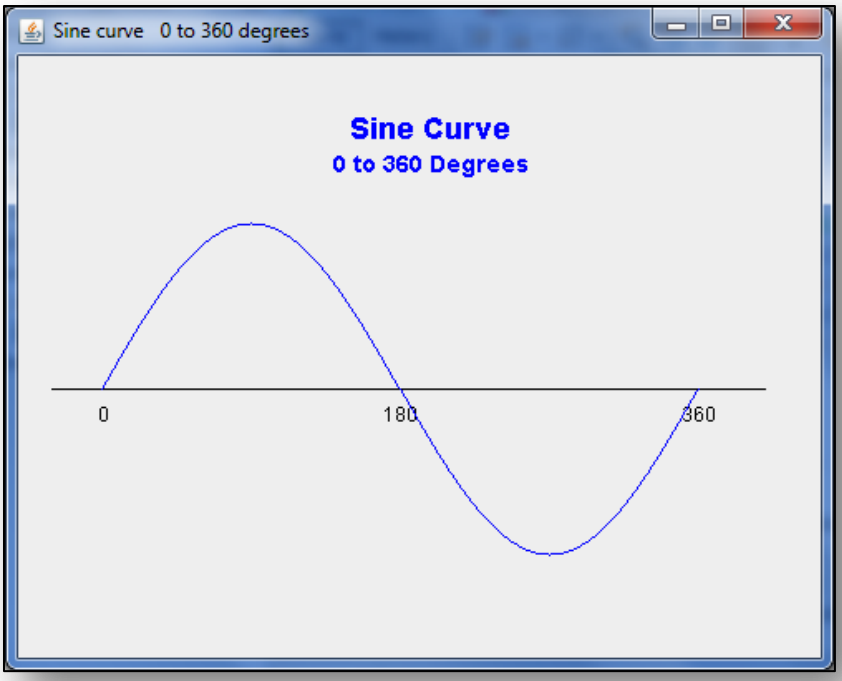
It is much easier to draw graphs of functions than it is to draw histograms or pie charts. We simply need to have a loop that goes through 360 degrees, calculates the function for each degree, then draws the function on the screen. However, some factors are necessary to convert the value of the function for each degree into coordinates on the screen.

For example, the sine of an angle varies from +1 to -1. If we wish to plot the point (x,y) where x is the angle and y is sine(x), we need to use a few factors to enlarge the graph the image on the screen.

The following pseudocode shows a short algorithm for plotting a sine curve. The value of the sine function is multiplied by 100, giving us the range -100 to +100, and it is added to 200 to position it on the screen:

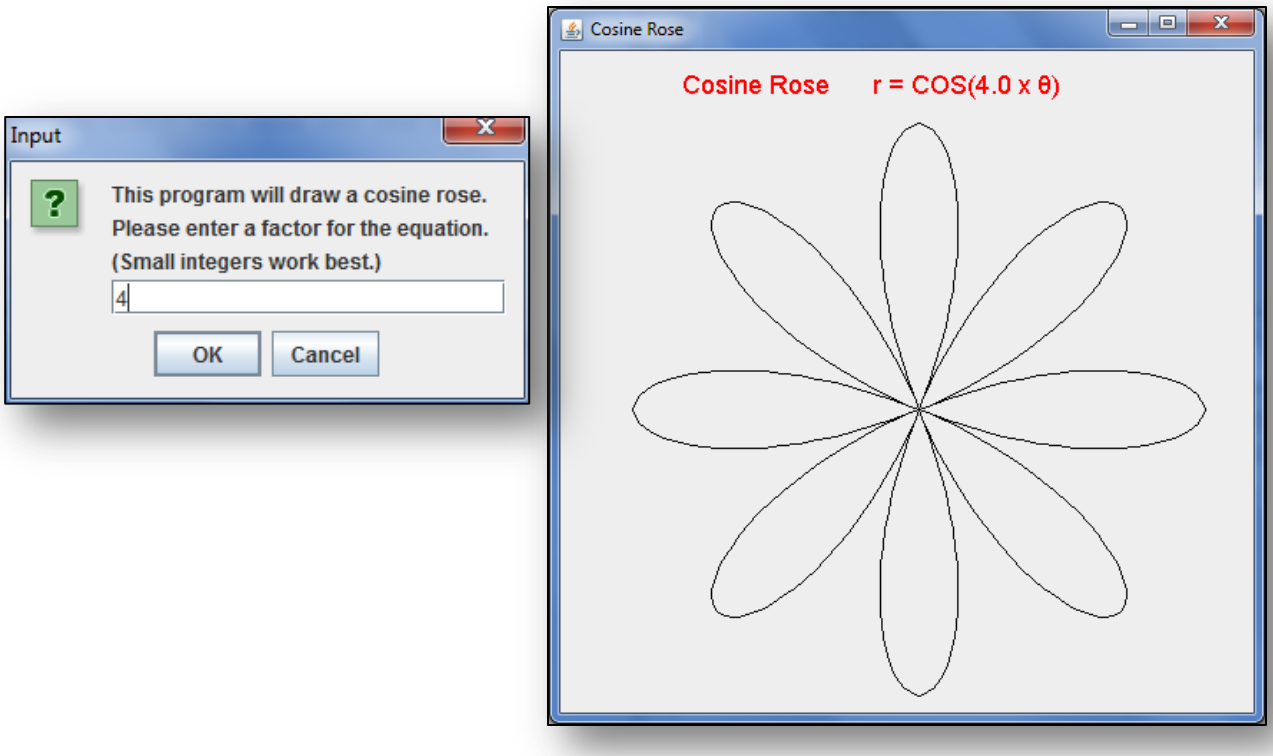
```
for(x =0.0; x <=360.0; x++)
    {
    y = 100.0 * Math.sin( Math.toRadians(x) );
    plot (x,y)
    }
```

The flowing image was drawn in a similar fashion, but instead of plotting each point, a line was drawn from each point on the curve to the next. The code for this is in the NetBeans project *SineCurve*, included with the files for this chapter. It uses several factors to line things up on the screen, and adds a few labels, as described in the comments in the source code.



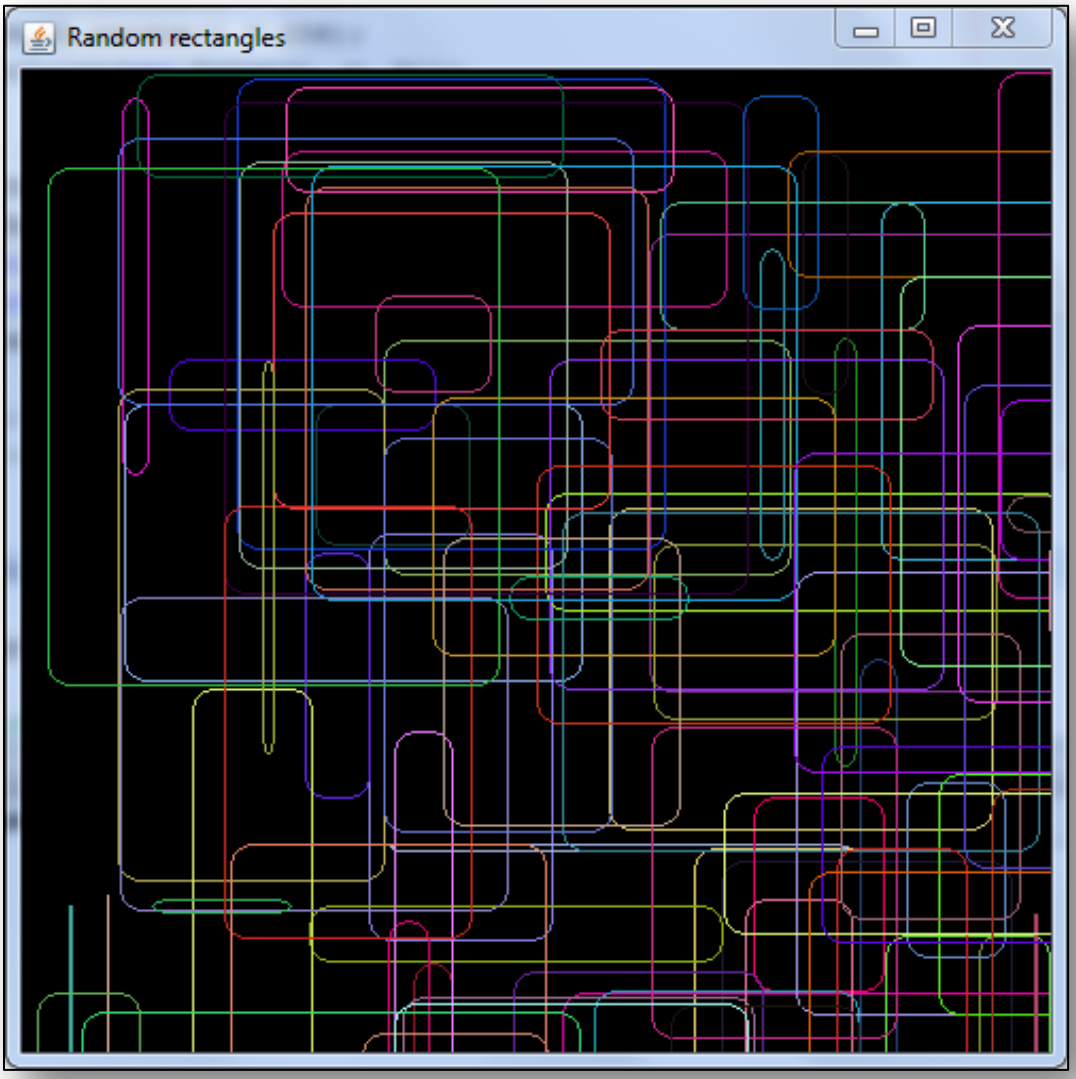
We can also convert the drawing to polar coordinates using parametric equations, and draw what is known as a cosine rose. Basically, a cosine curve is wrapped around a circle.

The image below shows this. The code for this is included in the Netbeans project *CosineRose*. It starts with a *JOptionPane* dialog box to ask the user for a factor that will alter the image. The code uses a time delay with try-catch blocks for exception handling, which is described in the next chapter.



11.7 Programming Example – Abstract Computer Graphics

Using the graphics commands shown in this chapter and a little creativity, it is fairly easy to create some original computer artwork. Such as the example below:



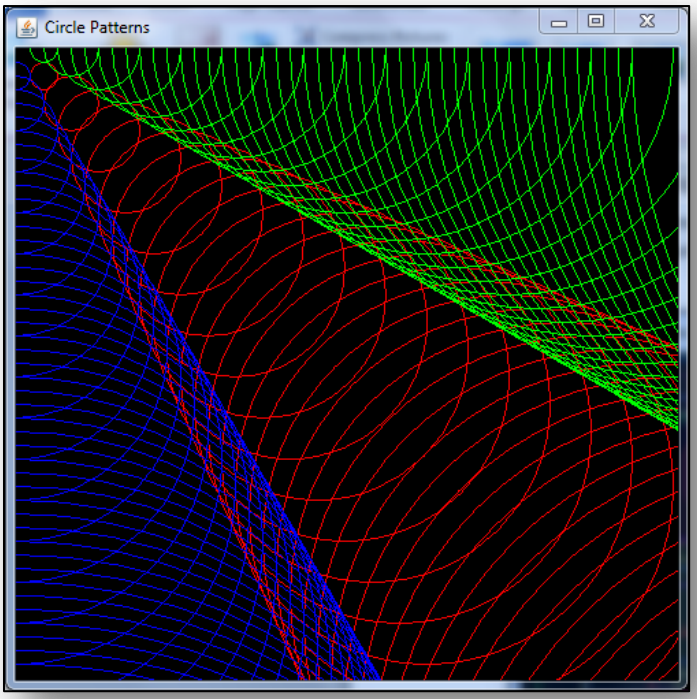
Figurative art or abstract art can be drawn using various graphics commands. Figurative art represents things form the physical world. Abstract art represents more abstract concepts. The *Random Rectangles* image above is abstract; The Bedroom Window below is figurative.

The art can also be dynamic, which means putting time delays in the program to allow the user to see the art being drawn or see the art changing while the program runs.

Some of the examples shown here are abstract, dynamic art. To see them in action run the program included in the NetBeans projects *RandomRectangles* and *CirclePatterns* included with this chapter.

Netbeans projects named *RandomRoses* and *RandomCards* are also included with the files for this chapter.

Warning: Some dynamic art programs flash on the screen and could cause problems for people prone to seizures, such as the *CirclePatterns* program.



Chapter Review

Section 1 of this chapter describes software and hardware used for computer graphics including what a graphics processing unit (GPU) is and why GPUs are used with modern computers.

Section 2 describes the use of the *java.awt.Canvas* and *java.awt.Graphics* classes

Section 3 is a long section **describing** how to work with of some of the most commonly used drawing methods in the Graphics class. It includes a discussion of inverted Cartesian coordinates.

Section 4 shows an example of the Graphics class to create a histogram. The process of laying out the design for the histogram can be a tedious process. so students are not expected to do this as part of this class.

Section 5 shows an example using the Graphics class to create a pie chart. The software included with this chapter has a program to draw a pie chart from a data file. Students can draw different pie charts by changing the data file.

Section 6 demonstrates how to use the Graphics class to graph a trigonometric function. It also has a sample program that draws a cosine rose.

Section 7 discusses using the Graphics class to create abstract and figurative computer artwork, including still images and dynamic art.

Chapter Questions

1. What is a GPU? Why is it used for computer graphics?
2. What is the Canvas class in Java? Why is it used to in conjunction with the Graphics class?
3. What is coordinate system is used for most computer graphics? How does it compare to a Cartesian coordinate system? Historically, how did this come to be used for computer graphics?
4. What coordinates must be specified to draw a rectangle using the drawrectangle() method?
5. How can we draw a filled rectangle using methods from the Java Graphics class?
6. How can we draw a circle using methods from the Java Graphics class? What about a triangle or hexagon?
7. What is the system of measurement and direction for drawing arcs of circle and ovals using the drawArc() method?
8. How can we draw text on a Canvas using methods from the Graphics class? How can we affect the appearance of the text?
9. What is the difference between figurative and abstract images?
10. What is the meaning of the term *dynamic art* with reference to computer generated images?

Chapter Exercise

Your task for this chapter is to write a Java program using the Java Graphics class to create an example of a computer generated image. This is an opportunity for you to explore computer graphics and exercise some individual creativity.

You should submit well-documented code, along with a lab report describing your work. What happened in the process of exploration and design for your project? How does your code work? How did you use other features of Java, such as branching routines or loops in your work? Tell us about the thoughts behind your finished work. What influenced your work?

Your work can be as simple or as complex as you would like, but be careful about taking on something that will be enormously time consuming. If you decide to work on figurative art, It might help to start with a single idea or concept, such as *snowfall in the city*, or *along the Schuylkill.* Figurative art can also be static or dynamic. Imagine how, for example, the *Bedroom Window* example at the end of this chapter could be subtly dynamic.

Your finished image for this project should be suitable for a general audience.

This assignment will be the final project for our class, although you will have a short assignment regarding Exceptions.

Contents