Chapter 3: Beginning graphics: Working with shapes

In this chapter, we'll see how to use Java to create graphical images using the Java2D packages. The earliest versions of Java allowed developers to create with relative ease programs with a significant graphical component. However, beginning with release 1.2, Java has incorporated sophisticated graphical tools that are somewhat similar to PostScript. Our discussion here aims to introduce a few of these tools. In the following chapters, we will see some ideas for adapting them to mathematical illustrations.

1: A first example

Let’s look at our first drawing program, which draws a simple rectangle. This example uses only graphics features available in Java 1.1, but it shows the outline of how we will draw and display illustrations.

```java
import javax.swing.*;
import java.awt.*;
public class DrawRectangle extends JPanel {
    public void paintComponent(Graphics g) {
        super.paintComponent(g); // always include this
        g.drawLine(100, 100, 100, 200);
        g.drawLine(100, 200, 300, 200);
        g.drawLine(300, 200, 300, 100);
        g.drawLine(300, 100, 100, 100);
    }
    public static void main(String[] args) {
        DrawRectangle drawRectangle = new DrawRectangle();
        JFrame frame = new JFrame("Draw Rectangle");
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.getContentPane().add(drawRectangle);
        frame.setSize(400, 300);
        frame.setVisible(true);
    }
}
```

Notice that a class called `DrawRectangle` is defined and that this class extends `JPanel`. You should think of a `JPanel` as being like the canvas on which we draw and then place into a frame. (In fact, earlier releases of Java used for these purposes a class called `Canvas`, an entirely appropriate name.) We are subclassing `JPanel` to customize the `paintComponent` method. This will be a common technique.

Let’s begin with a quick tour through the `main` method. To begin, a `DrawRectangle` is instantiated to create the picture we are drawing. Then a `JFrame` is instantiated and the `DrawRectangle` added to the `JFrame`. A `JFrame` essentially is the window seen on the computer screen that will hold the figure. After setting its size, the `JFrame` is displayed on the computer screen with the method `setVisible(true)`. For now, you should just think that a drawing surface has been inserted into a window and the window displayed.
Let's now focus on the method \texttt{paintComponent} of \texttt{DrawRectangle} for this is where the drawing instructions are given. When the \texttt{JFrame} is displayed on the computer screen, the \texttt{paintComponent} method of anything added to it is invoked. The method is called with a \texttt{Graphics} argument, an important class providing a \textit{rendering engine}. Simply said, a rendering engine, in this case the \texttt{Graphics} object, translates between the graphical commands we state and the actual picture that is displayed on the computer screen. Here, we ask to draw four lines using \texttt{Graphics}'s \texttt{drawLine} method. We do not need to worry about the details of how the image will be displayed.

To draw a line between two points \((x_1, y_1)\) and \((x_2, y_2)\), we use the method

\begin{verbatim}
g.drawLine(x1, y1, x2, y2);\end{verbatim}

The arguments given here are \texttt{ints} and may be thought of as specifying pixels within the \texttt{JFrame}.

Points are described in a coordinate system whose origin is at the upper left corner of the window. The first coordinate increases as we move to the right and the second coordinate increases as we move down the window. This choice of coordinate system requires a bit of consideration: for instance, since it is a left-handed coordinate system, angles are measured positively in the \textit{clockwise} direction. Earlier versions of Java did not allow the programmer to change the coordinate system. However, we will see that Java2D allows us to modify the coordinate system by an affine transformation in the same way this is done in PostScript.

Finally, the instruction \texttt{super.paintComponent(g)} should always be the first instruction in the \texttt{paintComponent} method of a \texttt{JPanel}. Without it, some unexpected, and very weird, behavior may result. The reason for this is a bit technical and not really relevant for us.

\textbf{Exercise 1:} Draw an equilateral triangle whose side is 100 pixels and with one vertex at \((100, 100)\).

\textbf{Exercise 2:} Draw a 3:4:5 right triangle whose hypotenuse is 100 pixels long and displayed horizontally.

\subsection*{2: Java2D}

With release 1.2, Java began providing a few packages that add significantly more powerful graphical capabilities. Grouped together, these packages form which is known as \texttt{Java2D}. In the next example, we will draw the following picture as a way of introducing \texttt{Java2D}. 

![Graphical diagram]
import javax.swing.*;
import java.awt.*;
import java.awt.geom.*;

public class Distance extends JPanel {
    public void paintComponent(Graphics gfx) {
        super.paintComponent(gfx);  // standard setup
        Graphics2D g = (Graphics2D) gfx;
        g.setRenderingHint(RenderingHints.KEY_ANTIALIASING,
                           RenderingHints.VALUE_ANTIALIAS_ON);
        // define the elements we will draw
        Point2D.Double p1 = new Point2D.Double(100, 100);
        Point2D.Double p2 = new Point2D.Double(200, 150);
        Line2D.Double line = new Line2D.Double(p1, p2);
        double r1 = 40; double r2 = 20;
        Ellipse2D.Double circle1 =
            new Ellipse2D.Double(p1.x - r1, p1.y - r1, 2*r1, 2*r1);
        Ellipse2D.Double circle2 =
            new Ellipse2D.Double(p2.x - r2, p2.y - r2, 2*r2, 2*r2);
        // draw line
        g.setPaint(Color.black);
        g.draw(line);
        // fill circles
        g.setPaint(new Color(51, 51, 204));
        g.fill(circle1); g.fill(circle2);
        // draw circles
        g.setPaint(Color.black);
        g.draw(circle1); g.draw(circle2);
    }

    public static void main(String[] args) {
        Distance distance = new Distance();
        distance.setBackground(Color.white);
        distance.setPreferredSize(new Dimension(250, 250));
        JFrame frame = new JFrame("Distance");
        frame.setDefaultCloseOperation(JFrame.DISPOSE_ON_CLOSE);
        frame.getContentPane().add(distance);
        frame.pack();
        frame.setVisible(true);
    }
}

This example is different, of course, from the first though the outline is similar. We define the
\texttt{paintComponent()} method of a subclass of JPanel to draw our image, which is displayed
inside a JFrame.

However, beginning with release 1.2, Java provides an instance of Graphics2D, a subclass of
Graphics, when the paintComponent method is called. This is a significantly more powerful rendering
engine and provides access to the features of the Java2D packages. Therefore, to use this rendering
engine, we need first to declare that it is an instance of Graphics2D, a feat accomplished by casting the Graphics object to a Graphics2D in the following way:

```java
Graphics2D g = (Graphics2D) gfx;
```

This will be a common instruction anytime you wish to use Java2D.

Next we set up the objects that we would like to draw. You will notice that these are defined in a relatively intuitive way. First, we define points using the java.awt.geom.Point2D class. This class is a convenient way to store the coordinates of a point and pass them to a variety of methods. It is not possible, however, to display an instance of Point2D. Next, we define a Line whose endpoints are the given points and two Ellipses, circles in this case, centered the two points. The Ellipse objects are defined in a slightly unusual way: the arguments given to the constructor describe the smallest rectangle that contains the ellipse by specifying its upper left corner, its width and its height.

The inclusion of Double in the name of the class Point2D.Double is due to the fact that we are representing the coordinates of a point using doubles. There is another version of this class, Point2D.Float, in which the coordinates are represented by floats, 32-bit floating point numbers. Of course, doubles give greater accuracy in computations, but floats will run faster. This may be an issue for consideration when constructing animations.

Now it’s time to do the drawing. A Graphics2D always has a java.awt.Paint object that is used when any drawing is performed. For the time being, you may think of a Paint object as being merely an instance of java.awt.Color though we will eventually see that it can do much more. A convenient class field is Color.black, and the instruction g.setFillPaint(Color.black) has the effect of loading our brush with black paint. The line is then drawn with the draw method, whose result is similar to the stroke command in PostScript.

Next we change the Paint on our brush by building a custom Color object. Color objects may be constructed by specifying the amount of red, green and blue (from the range 0 - 255) used to create the Color. After this, we fill the circles. This has the effect of shading the inside of the circles. Then set the Paint back to Color.black and draw the outline of the circles using the draw method.

If you study the Java API, you will see that the methods draw and fill in Graphics2D require an argument of type Shape. The classes we have seen here, Line2D and Ellipse2D, subclass Shape and so are acceptable arguments for these methods. (Technically, Shape is an interface, a type of gadget we will meet later on.)

**Exercise 3:** What happens if you fill a line?

**Exercise 4:** How is the image changed if you fill and draw the circles first and then draw the line? Explain this behavior.

**Exercise 5:** Create a Java program to illustrate the Proposition I from Book I of *The Elements*. In particular, you should specify two initial points, shown below in blue, and construct an equilateral triangle containing the two initial points as vertices. Shade the third vertex red.
Exercise 6: Create a class `GraphicalPoint` that will display points like those shown in the figure above. You should specify the coordinates \((x, y)\), the `Color` and the size and provide a method for displaying the point using methods like this:

```java
public class GraphicalPoint {
    double x, y;
    double size;
    Color color;
    public GraphicalPoint(double x, double y);
    public setPaint(Color color);
    public setSize(double size);
    public void plot(Graphics2D g);
}
```

Exercise 7: Enhance the `GraphicalPoint` class to allow the point to be drawn as either a filled circle or filled square. Add the following fields and instance method:

```java
public static int CIRCLE;
public static int SQUARE;
public int style;
public void setStyle(int style);
```

3: Some more graphics primitives

In the last section, we met a few objects, namely lines and circles, that may be displayed graphically. We call such an object a *graphical primitive*. We will now meet a few more.

Besides lines, Java2D also gives us the ability to work with quadratic and cubic curves.

- A quadratic curve is defined by three control points, \(P_0\), \(P_1\) and \(P_2\), as shown below:
This is a quadratic curve of the form

\[ \gamma(t) = a + bt + ct^2 \text{ where } 0 \leq t \leq 1 \]

and

\[
\begin{align*}
\gamma(0) &= P_0 \\
\gamma'(0) &= 2(P_0 - P_1) \\
\gamma(1) &= P_2
\end{align*}
\]

A quadratic curve may be instantiated as

```java
QuadCurve2D.Double quad = new QuadCurve2D.Double(double x1, double y1, double x2, double y2, double x3, double y3);
```

or by using

```java
QuadCurve2D.Double quad = new QuadCurve2D.Double();
quad.setCurve(Point2D p1, Point2D p2, Point2D p3);
```

**Exercise 8:** Create a Java program to illustrate the figure above.

**Exercise 9:** Quadratic curves may be obtained through an elegant subdivision process. Notice that if we consider the curve \( \gamma(t) \) with either \( 0 \leq t \leq 1/2 \) or \( 1/2 \leq t \leq 1 \), we are left with two new quadratic curves, the control points of which are easily determined from the control points of the original curve. If we define derived points by

\[
\begin{align*}
Q_1 &= (P_0 + P_1)/2 \\
Q_2 &= (P_1 + P_2)/2 \\
Q_3 &= (Q_1' + Q_2')/2,
\end{align*}
\]

then the first curve is given by the control points \( P_0, Q_1 \) and \( Q_3 \) while the second curve is given by \( Q_3, Q_2 \) and \( P_2 \). This is shown in the figure below. Create a program that takes three control points, finds the derived control points and then draws the two resulting quadratic curves.
Exercise 10: Create a program that draws an approximation to a quadratic curve by drawing straight lines obtained through continual subdivision using a command-line parameter $n$. You should begin with three control points and create the control points for the two quadratic curves that describe the two halves of the original quadratic curve. Continue this process of subdivision $n$ times so that $2^n$ quadratic curves are obtained. Then replace the curves by straight lines. This may be nicely implemented recursively.

Exercise 11: Repeat the exercise above with the following modification. Subdivide the original curve until the curves obtained are sufficiently small, say when all control points lie within one pixel of one another, and then draw straight lines.

- A cubic curve is defined by four control points, $P_0, P_1, P_2$ and $P_3$, as shown below:

This curve has the form

$$\gamma(t) = a + bt + ct^2 + dt^3$$

where

- $\gamma(0) = P_0$
- $\gamma'(0) = 3(P_0 - P_1)$
- $\gamma'(1) = 3(P_3 - P_2)$
- $\gamma(1) = P_3$
The first and fourth control points determine the endpoints of the curve and the second and third determine the velocity of the cubic curve at the endpoints.

A cubic curve may be instantiated as

```
CubicCurve2D.Double cubic = new CubicCurve2D.Double(double x1, double y1,
          double x2, double y2, double x3, double y3, double x4, double y4);
```

or using

```
CubicCurve2D.Double cubic = new CubicCurve2D.Double();
quad.setCurve(Point2D p1, Point2D p2, Point2D p3, Point2D p4);
```

**Exercise 12:** Reproduce the figure above.

**Exercise 13:** There is no way for a computer to render a circle exactly. PostScript chooses to approximate a circle by a sequence of cubic curves. Construct an approximation to a circle by drawing four congruent cubic curves. The ratio of the length of the segment joining an endpoint and its corresponding control point to the radius of the circle should be 0.552.

Cubic curves may be drawn by a subdivision process similar to that for quadratic curves.

Several useful classes are subclasses of RectangularShape. In general, a RectangularShape is described by specifying the upper left coordinates of the rectangle along with its width and height.

- We have already met Ellipse2D. In general, it is constructed as

```
Ellipse2D.Double ellipse = new Ellipse2D.Double(double ulx, double uly,
               double width, double height);
```

Such an ellipse will always be oriented with its axes parallel to sides of the frame in which it resides. However, we will see in the next chapter how to rotate an Ellipse.

- A simple Rectangle2D is instantiated in the same way:

```
Rectangle2D.Double ellipse = new Rectangle2D.Double(double ulx, double uly,
               double width, double height);
```

- An arc of an ellipse is described by Arc2D and may be instantiated using

```
Arc2D.Double arc = new Arc2D.Double(double ulx, double uly, double width, double height,
               double angleStart, double angleEnd, int closure);
```

The first four arguments define the bounding box of the ellipse while the next two, angleStart and angleEnd, define the angular portion of the ellipse we would like to work with. That is, if we parametrize the ellipse as

\[ \gamma(t) = (a \cos t, b \sin t), \]

then we look at the portion of the ellipse where \( \text{angleStart} \leq t \leq \text{angleEnd} \). Please note that these angles, for historical reasons, are measured in degrees. The method Math.toDegrees may be used to convert an angle measure from radians into degrees.
The final argument, closure, determines how the arc is completed. It may take on one of three values, `Arc2D.OPEN`, `Arc2D.PIE` and `Arc2D.CHORD`, with the results shown below.

![Arc examples](image)

**Exercise 14:** The following figure gives a visual explanation of the two sums:

\[
1 + 2 + 3 + 4 + \ldots + n = \frac{n(n + 1)}{2}
\]

\[
1^3 + 2^3 + 3^3 + 4^3 + \ldots + n^3 = \left[ \frac{n(n + 1)}{2} \right]^2
\]

Create a Java program that draws it.

**Exercise 15:** Begin by specifying the three vertices of a triangle and draw the inscribed circle along with the three angle bisectors and angle markers to indicate the angle bisectors as shown below.

![Triangle with inscribed circle and angle bisectors](image)
Here are a few things to remember: the coordinate system is left-handed so angles are measured in the clockwise direction, the angles in the Arc2D constructor are measured in degrees and the angle returned by Math.atan2(y, x) is measured in radians. In the next chapter, we will see how to alter the coordinate system.

4: Creating more general shapes

In the last section, we saw a few simple graphical primitives. Now we’ll see a class, GeneralPath, that allows us to put them together easily. This class allows a Java programmer to create shapes in a style similar to that used in PostScript.

To instantiate a GeneralPath, we say simply

```java
GeneralPath path = new GeneralPath();
```

and then add pieces—lines, quadratic curves and cubic curves—to the path. For instance, the following code generates the figure below.

```java
g.setPaint(Color.black);
GeneralPath path = new GeneralPath();
path.moveTo(50, 50);
path.lineTo(150, 150);
path.quadTo(200, 200, 250, 150);
path.curveTo(250, 250, 150, 250, 150, 200);
path.closePath();
g.draw(path);
```

Please note that these methods require floats. If you are computing with doubles, you will therefore need to cast the coordinates to floats.

Besides adding line segments, quadratic curves and cubics to an existing path, one may add a general Shape using the append method of GeneralPath.
**Exercise 16:** Draw a circle of radius 100 pixels and add inscribed \( n \)-gons where \( n = 4, 8 \) and 16.

---

**5: Displaying text**

Since we’ve been discussing how to draw and fill simple shapes, it may seem something of a departure now to discuss how to display text. In fact, we will see later in these notes that text is actually very much like the Shapes we have been considering and can be worked with in a similar way. However, since text operations are so common, Java provides some simple ways to deal with it.

First, a Graphics2D object stores a Font that is used for displaying text. Java2D supports relatively sophisticated uses of fonts. For now, however, it will suffice to work with a relatively simple set of fonts provided by Java2D. To instantiate a Font, we give its family, style and size as

```java
new Font("sanserif", Font.BOLD, 12);
```

This gives us a standard bold Font without serifs at 12 points. We then set the Font of our Graphics2D:

```java
g.setFont(new Font("sanserif", Font.BOLD, 12));
```

To display the word "text" at a location \((x, y)\), we may use the method

```java
g.drawString("text", x, y);
```

where \( x \) and \( y \) are floats.

It can be important to recognize how the point \((x, y)\) determines where the String appears. Consider the following figure.

The point \((x, y)\) given to the drawString method is shown in red. This point determines the baseline of the String as it appears on the screen.

There is a class, FontMetrics, that gives information helpful for measuring the layout of text. After the Font has been set, obtain a FontMetrics from the Graphics2D by

```java
FontMetrics fm = g.getFontMetrics();
```
A convenient way to obtain information about the size of the String we are to display is through the `getStringBounds` method:

```java
Rectangle2D rect = fm.getStringBounds("text", g);
```

This gives a rectangle that contains the String were it to be rendered at the point $(0, 0)$.

**Exercise 17:** Explain why `rect.y` is typically negative and `rect.height + rect.y` is positive.

**Exercise 18:** Construct the figure shown above. The rectangle containing the text should be centered in the JPanel.

**Exercise 19:** Construct a method

```java
public void drawString(String s, Graphic2D g, double x, double y, int anchor)
```

that draws the String $s$ in a box specified by the point $(x, y)$ and the argument anchor. The value of anchor should come from a list of directions such as EAST, NORTHEAST, NORTH, CENTER and so on. For instance, if NORTHEAST is used, the point $(x, y)$ should be the upper right corner of the bounding box as shown above.

**Exercise 20:** Draw the picture of the coordinate system shown at the beginning of this chapter.

### 6: Iterating over a path

The `PathIterator` class is useful for determining the constituent pieces—lines as well as quadratic and cubic curves—of a `Shape`. This is similar to the `pathforall` command in PostScript.

A `PathIterator` object is obtained by calling the `getPathIterator` method of the `Shape` in question:

```java
PathIterator iterator = shape.getPathIterator(null);
```

(The argument is meant to be an `AffineTransform`, a class we will meet in the next chapter.)

We can then step through the pieces of the `PathIterator` using the `currentSegment` method:

```java
public int currentSegment(float[] coords)
```

The argument is an array of six floats that describe the geometry of the current piece of the `PathIterator`. The int returned tells us what type of segment—line, quadratic or cubic curve—we are working with. To obtain the next segment, use the `next` method of the `PathIterator`.

Here is some code that demonstrates how Java draws a circle. Line segments are drawn in red, quadratic curves in green and cubic curves in blue.
Ellipse2D.Double circle = new Ellipse2D.Double(50, 50, 200, 200);
PathIterator iterator = circle.getPathIterator(null);
double lastX=0, lastY=0, beginX=0, beginY=0;
while(!iterator.isDone()) {
    float[] coords = new float[6];
    int type = iterator.currentSegment(coords);
    switch(type) {
    case(PathIterator.SEG_MOVETO): {
        beginX = coords[0]; beginY = coords[1];
        lastX = coords[0]; lastY = coords[1];
        break;
    }
    case(PathIterator.SEG_LINETO): {
        Line2D.Double line = new Line2D.Double(lastX, lastY,
                                                coords[0], coords[1]);
        g.setPaint(Color.red);
        g.draw(line);
        drawPoint(lastX, lastY, g);
        lastX = coords[0]; lastY = coords[1];
        break;
    }
    case(PathIterator.SEG_QUADTO): {
        QuadCurve2D.Double quad =
                        new QuadCurve2D.Double(lastX, lastY,
                                                coords[0], coords[1],
                                                coords[2], coords[3]);
        g.setPaint(Color.green);
        g.draw(quad);
        drawLine(lastX, lastY, coords[0], coords[1], g);
        drawLine(coords[0], coords[1], coords[2], coords[3], g);
        drawPoint(lastX, lastY, g);
        drawPoint(coords[0], coords[1], g);
        lastX = coords[2]; lastY = coords[3];
        break;
    }
    case(PathIterator.SEG_CUBICTO): {
        CubicCurve2D.Double cubic =
                        new CubicCurve2D.Double(lastX, lastY,
                                                coords[0], coords[1],
                                                coords[2], coords[3],
                                                coords[4], coords[5]);
        ...
        break;
    }
    case(PathIterator.SEG_CLOSE): {
        ...
        break;
    }
    }
}
iterator.next();
The resulting figure shows that circles are drawn using four cubic curves as mentioned in an earlier exercise.

We mentioned earlier that text may be treated as a **Shape**. Here is how to do it using some classes from **java.awt.font**.

```java
Graphics2D g = (Graphics2D) gfx;
String string = "g";
Font font = new Font("serif", Font.PLAIN, 250);
g.setFont(font);
FontRenderContext frc = g.getFontRenderContext();
GlyphVector glyphVector = font.createGlyphVector(frc, string);
Shape shape = glyphVector.getGlyphOutline(0, 75f, 200f);
```

In this way, we can see that the character "g" is created using only quadratic curves and line segments in this font.
Another class, FlatteningPathIterator, provides a PathIterator that approximates a given PathIterator and in which all segments are line segments. The maximum error allowed in the approximation is controlled by an argument to the constructor of the FlatteningPathIterator.

Shown below is the result of using the FlatteningPathIterator to display the image of the character “g” under the complex function $f(z) = 1/z$. 