Section B – Java 2D

20 Graphics2D: Introduction

Graphics evolved and developed within Java initially through the Abstract Window Toolkit, which was extended to include swing, shortly followed by Java 2D and then finally Java 3D.

Abstract Window Toolkit

The Abstract Window Toolkit (AWT) which is part of the Java Foundation Classes (JFC) was developed to support Graphical User Interface (GUI) programming for windows applications and applets.

A number of graphics and imaging tools, including shape, color, and font classes were also incorporated.

Swing

Swing which is also part of the Java Foundation Classes (JFC) introduced to further enhance support for and extend the Graphical User Interface (GUI) components with a pluggable look and feel (Macintosh, Microsoft, Solaris).

Swing improved numerous AWT components such as **Button**, **Scrollbar**, **Label**, etc. which became **JButton**, **JScrollbar**, **JLabel** respectively. New components were also introduced e.g. tree view, list box, and tabbed panes etc.

Java 2D

The Java 2D API is covered in the j2sdk documentation, which should be installed in your j2sdk directory. The main resources for 2D comprise of Sun's Java Tutorial - Trail: 2DGraphics and the Programmer's Guide to the JavaTM 2D API although it will also be discussed here. Other tutorials include The Java 2D API from the 1998 Java One conference, Java2D: An Introduction and Tutorial, Johns Hopkins University. Also the Sun's Java 2D API FAQ's.

The key packages for enabling graphics (including line drawing, text and images) are from the Abstract Windows Toolkit (awt) package:

- **java.awt.image**
- **java.awt.color**
- **java.awt.font**
- **java.awt.geom**
- **java.awt.print**
- **java.awt.image.renderable**

To be able to draw/render onto components, an application will use the **Graphics** Class, for all graphics contexts. To enable this the **paint()** method is called by Java when an application needs to draw/render itself to the screen. The **paint()** method has one parameter/argument, a **Graphics** object (sometimes called a graphics context) that is defined by the **Graphics** class, imported from **java.awt.Graphics**.

Graphics methods for drawing geometric primatives

- **draw3DRect()**
- **drawArc()**
- **drawLine()**
- **drawOval()**
- **drawPolygon()**
- **drawPolyline()**
- **drawRect()**
- **drawRoundRect()**
- **drawString()**

```
draw3DRect(int x, int y, int width, int height, boolean raised)
drawArc(int x, int y, int width, int height, int startAngle, int arcAngle)
drawLine(int x1, int y1, int x2, int y2)
drawOval(int x, int y, int width, int height)
drawPolygon(int[] xPoints, int[] yPoints, int nPoints)
drawPolyline(int[] xPoints, int[] yPoints, int nPoints)
drawRect(int x, int y, int width, int height)
drawRoundRect(int x, int y, int width, int height, int arcWidth, int arcHeight)
drawString(String str, int x, int y)
```

The **Graphics2D** class extends the **Graphics** class enabling greater functionality for geometry, coordinate transformations, colour management and text layout.

```
java.lang.Object
  |
  +--java.awt.Graphics
        |
        +--java.awt.Graphics2D
```

The **Graphics2D** class inherits from the **Graphics** class and in a java program the **Graphics** object is cast to a **Graphics2D** object as follows:

```
paint(Graphics g)
{
   Graphics2D g2D = (Graphics2D) g;
    ...............
```

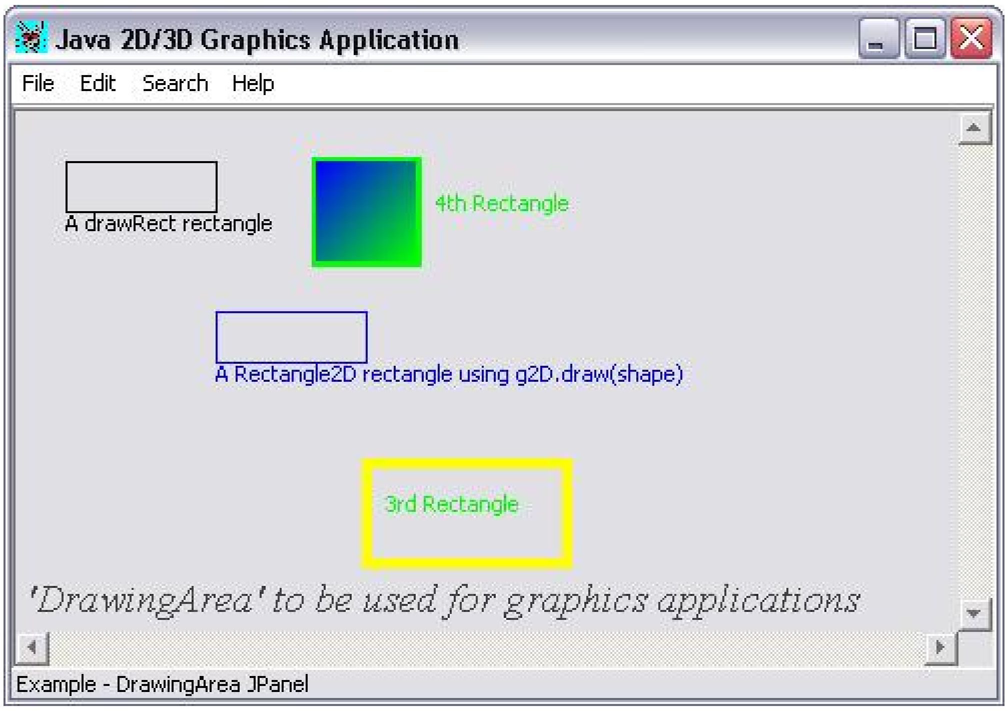
21 Graphics2D: Shapes

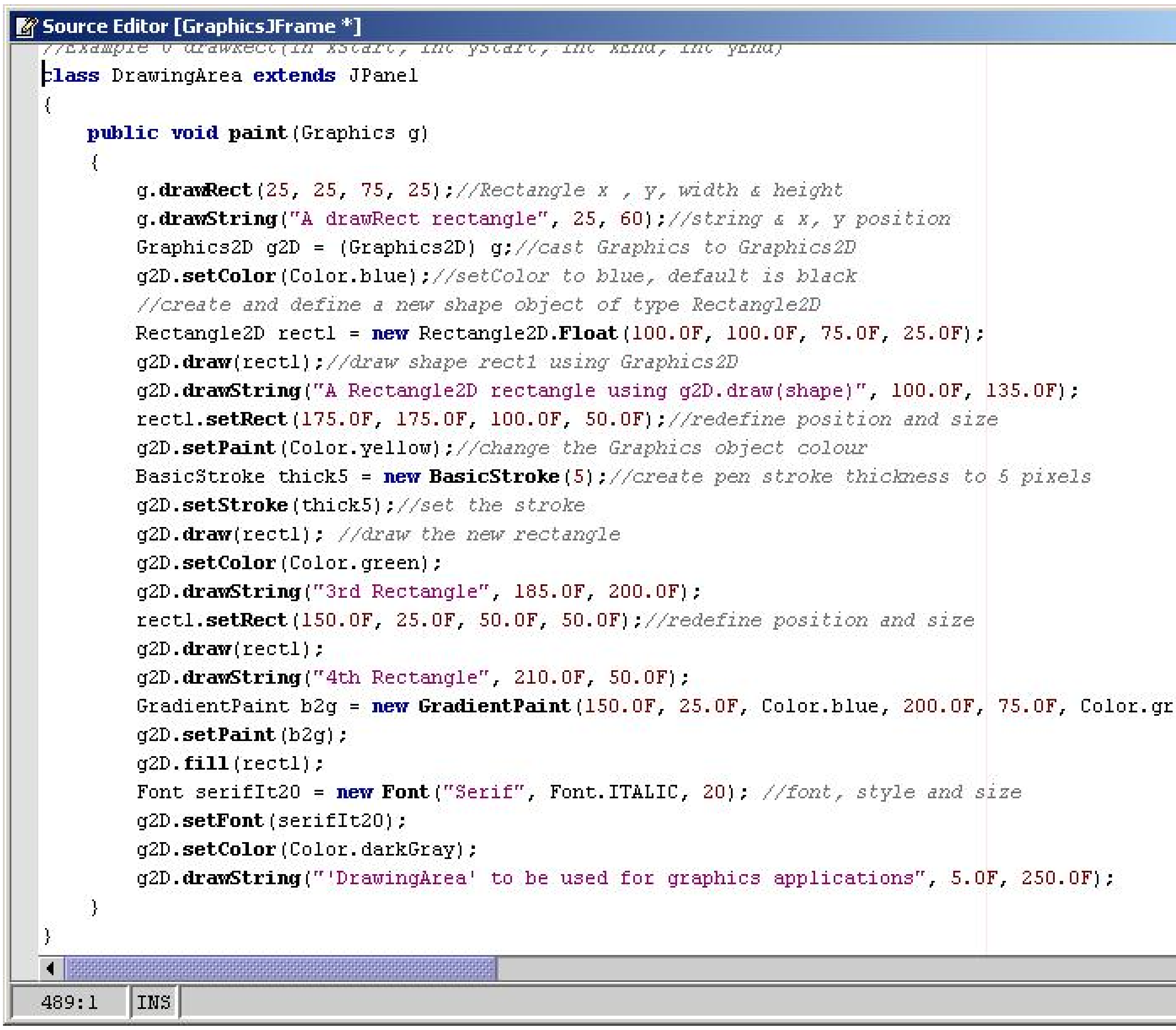
The key classes that enable the drawing/rendering of shapes are contained within in the **java.awt.geom** package as follows:

- Arc2D
- Area
- CubicCurve2D
- Dimension2D
- Ellipse2D
- GeneralPath
- Line2D
- Point2D
- QuadCurve2D
- Rectangle2D
- RectangularShape
- RoundRectangle2D

These classes allow the creation of most geometric shapes, which can then be drawn/rendered using **Graphics2D** by calling the **draw()**/ **fill()** methods. Java 2D allows the specifcation of coordinates using floating-point numbers and supports integer, double, and floating arithmetic in many places.

The example below (and subsequent exercises) will illustrate the use of some of classes available i.e. **Ellipse2D**, **Line2D**, **Point2D**, **Rectangle2D**.


Java 2D/3D Graphics Application
File
Edit
Search
Help
A drawRect rectangle
4th Rectangle
A Rectangle2D rectangle using g2D.draw(shape)
3rd Rectangle
'DrawingArea' to be used for graphics applications
Example - DrawingArea JPanel


Source Editor [GraphicsJFrame *]

```
//Example 6 drawRect(in xStart, int yStart, int xEnd, int yEnd)
class DrawingArea extends JPanel
{
    public void paint(Graphics g)
    {
        g.drawRect(25, 25, 75, 25);//Rectangle x , y, width & height
        g.drawString("A drawRect rectangle", 25, 60);//string & x, y position
        Graphics2D g2D = (Graphics2D) g;//cast Graphics to Graphics2D
        g2D.setColor(Color.blue);//setColor to blue, default is black
        //create and define a new shape object of type Rectangle2D
        Rectangle2D rect1 = new Rectangle2D.Float(100.0F, 100.0F, 75.0F, 25.0F);
        g2D.draw(rect1);//draw shape rect1 using Graphics2D
        g2D.drawString("A Rectangle2D rectangle using g2D.draw(shape)", 100.0F, 135.0F);
        rect1.setRect(175.0F, 175.0F, 100.0F, 50.0F);//redefine position and size
        g2D.setPaint(Color.yellow);//change the Graphics object colour
        BasicStroke thick5 = new BasicStroke(5);//create pen stroke thickness to 5 pixels
        g2D.setStroke(thick5);//set the stroke
        g2D.draw(rect1); //draw the new rectangle
        g2D.setColor(Color.green);
        g2D.drawString("3rd Rectangle", 185.0F, 200.0F);
        rect1.setRect(150.0F, 25.0F, 50.0F, 50.0F);//redefine position and size
        g2D.draw(rect1);
        g2D.drawString("4th Rectangle", 210.0F, 50.0F);
        GradientPaint b2g = new GradientPaint(150.0F, 25.0F, Color.blue, 200.0F, 75.0F, Color.gr
        g2D.setPaint(b2g);
        g2D.fill(rect1);
        Font serifIt20 = new Font("Serif", Font.ITALIC, 20); //font, style and size
        g2D.setFont(serifIt20);
        g2D.setColor(Color.darkGray);
        g2D.drawString("'DrawingArea' to be used for graphics applications", 5.0F, 250.0F);
    }
}
```

489:1 INS

Example 2D.1a: 2D circle & square.

Draw and label a circle and square to look like the one below using the **Graphics** drawX() methods, attempt to use similar screen positioning.

Hint: The objects circle and square do not exist in java, but ellipse and rectangle do!

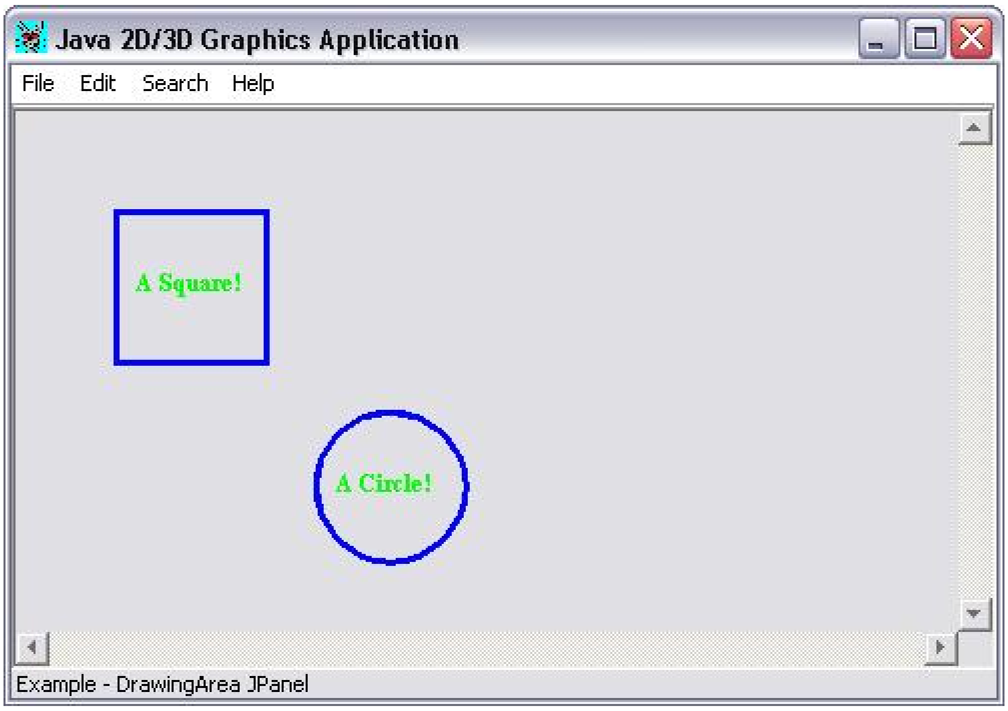


Figure 2D.1: 2D circle & square.

Example 2D.1b: 2D circle & square.

Adapt 1a above, to draw and label a circle and square using the **Graphics2D draw()** method, use the **java.awt.geom.*** shapes. **java.awt.geom.*** will also need to be imported.

Example 2D.1c: 2D java.awt.geom.*.

Use the **Graphics2D draw()** & **fill()** methods on some of the remaining **java.awt.geom.*** shapes.

Example 2D.2: 2D squares within a square.

Using **Line2D.Float** and **Point2D.Float**, draw a series of 7 squares within a square to look like the one below.

Each of the squares should be drawn from the mid point of the side of the previous square.

The original co-ordinates for rectangle should be (50.0F, 50.0F) (150.0F, 50.0F) (150.0F, 50.0F) (50.0F, 150.0F).

Use **System.out.println** to check the calculations for the new points, similar to:

```
Point 0 x1   50.0 x2 150.0 Xmidpoint 100.0 Y1   50.0 Y2   50.0 Ymidpoint  50.0
Point 1 x1 150.0 x2 150.0 Xmidpoint 150.0 Y1   50.0 Y2 150.0 Ymidpoint 100.0
Point 2 x1 150.0 x2   50.0 Xmidpoint 100.0 Y1 150.0 Y2 150.0 Ymidpoint 150.0
Point 3 x1   50.0 x2 100.0 Xmidpoint   75.0 Y1 150.0 Y2   50.0 Ymidpoint 100.0

New Point 0 Xmidpoint 100.0 Ymidpoint   50.0
New Point 1 Xmidpoint 150.0 Ymidpoint 100.0
New Point 2 Xmidpoint 100.0 Ymidpoint 150.0
New Point 3 Xmidpoint  75.0 Ymidpoint 100.0
```

Hint: Methods that will be needed include **setLine()**, **setLocation()** for **Line2D** and **getX()/getY()** for **Point2D**. A 'midPoints' function should be used to determine the mid-point co-ordinate of each side. The function will be used recursively for the 7 iterations.

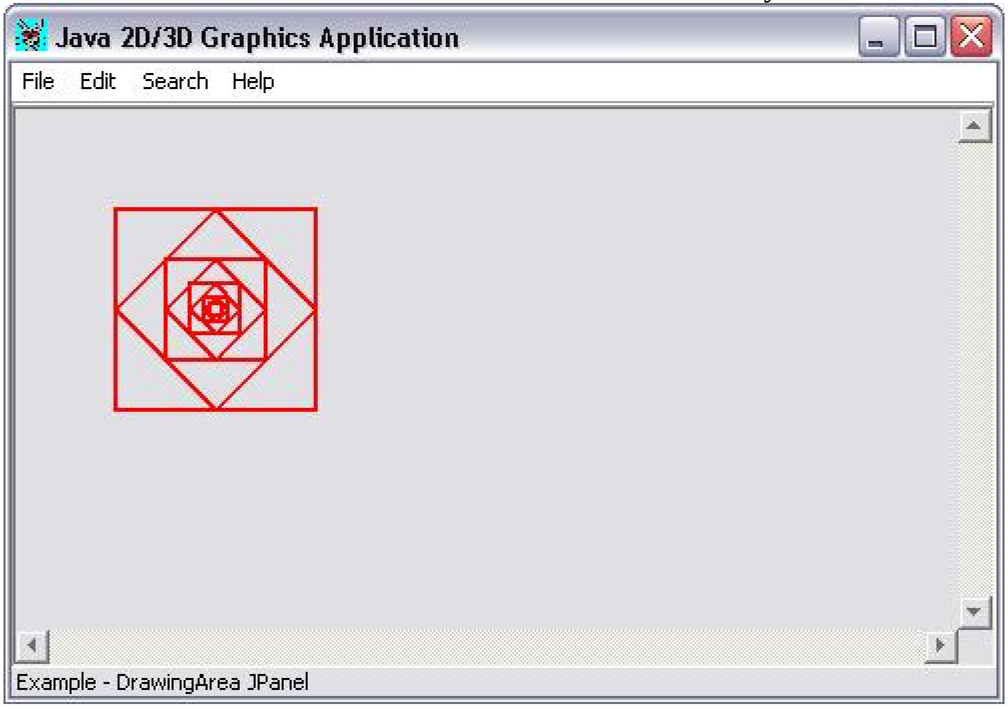


Figure 2D.2: 2D squares within a square.

Example 2D.3: Triangles within a triangle

Adapt the application developed in Exercise 2 above to draw a series of 4 triangles within a triangle to look like the one below.

Each of the triangle should be drawn from the mid point of the side of the previous triangle.

The original co-ordinates for triangle should be (50.0F, 50.0F) (250.0F, 50.0F) (150.0F, 250.0F).

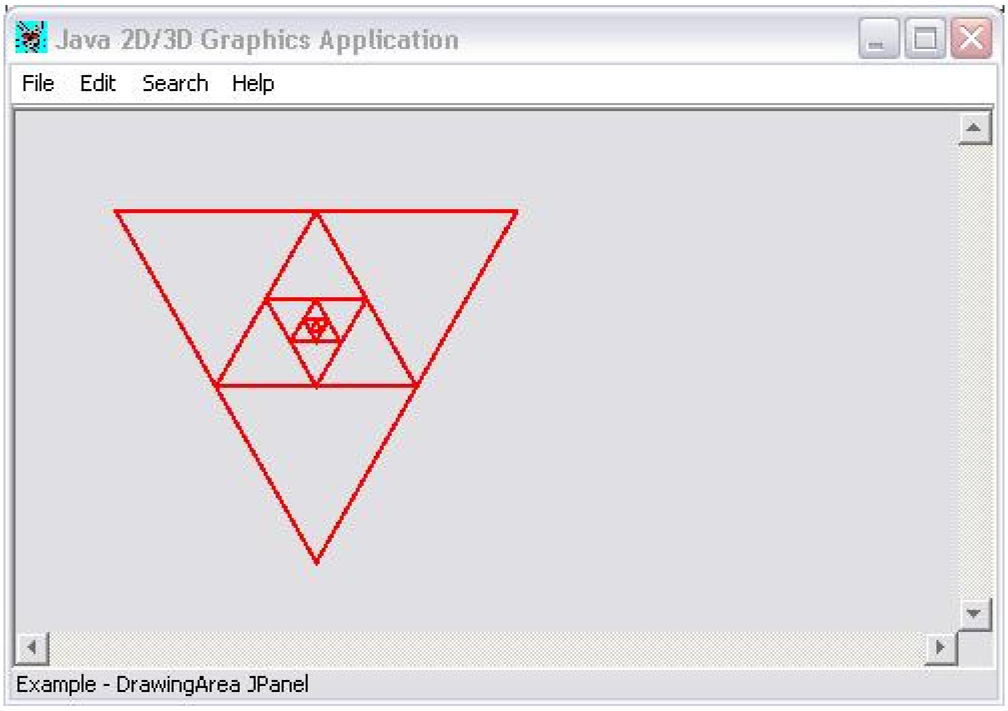
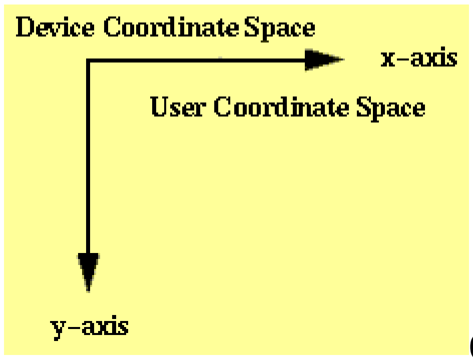


Figure 2D.3: 2D Triangles wiithin a triangle.

22 Graphics 2D: Coordinates

There are two coordinate spaces used in java graphics. These are Device Coordinate Space and User Coordinate Space.

Device Space and User Space coordinate systems are initially the same, with the origin (x=0, y=0) in the top-left corner of the screen.

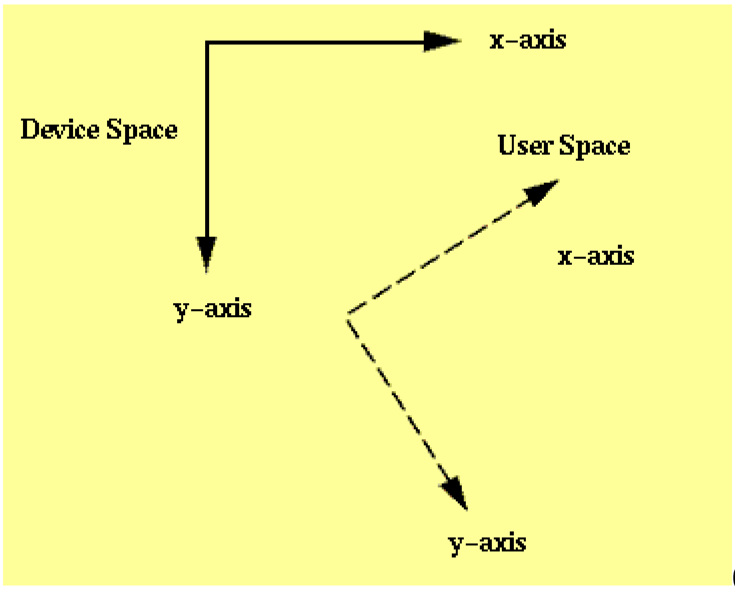


(javaworld (1998))

The coordinates are represented by whole (integer) pixels where a positive x value is to the right of the origin and a positive y value is moves down.

The range of x and y available will depend on the resolution of the screen being used. Most commonly this will be 1024 * 768 pixels.

The Device Space is that area of the screen onto which the graphics will be rendered on the screen. Whereas, UserSpace can be modified by a transform and/or a rotation (see AffineTransforms later).



(javaworld (1998))

Example 2D.4: Squares within a square (variation).

Adapt the application developed in Exercise 2 above to draw a series of 30 squares within a square to look like the one below.

Each of the square should be drawn offset from the previous square.

The initial square should be 200 by 200.

Hint: If the coordinate of a vertex a (say the top left hand corner) is (aX, aY) the new vertex (a1X, a1Y) to create the next square would be as follows:

```
  a1X = ((1- u) * aX) + (u*bX)
  a1Y = ((1- u) * aY) + (u*bY)
```

Where b is the vertex for the top right hand corner. Try u = 0.1.

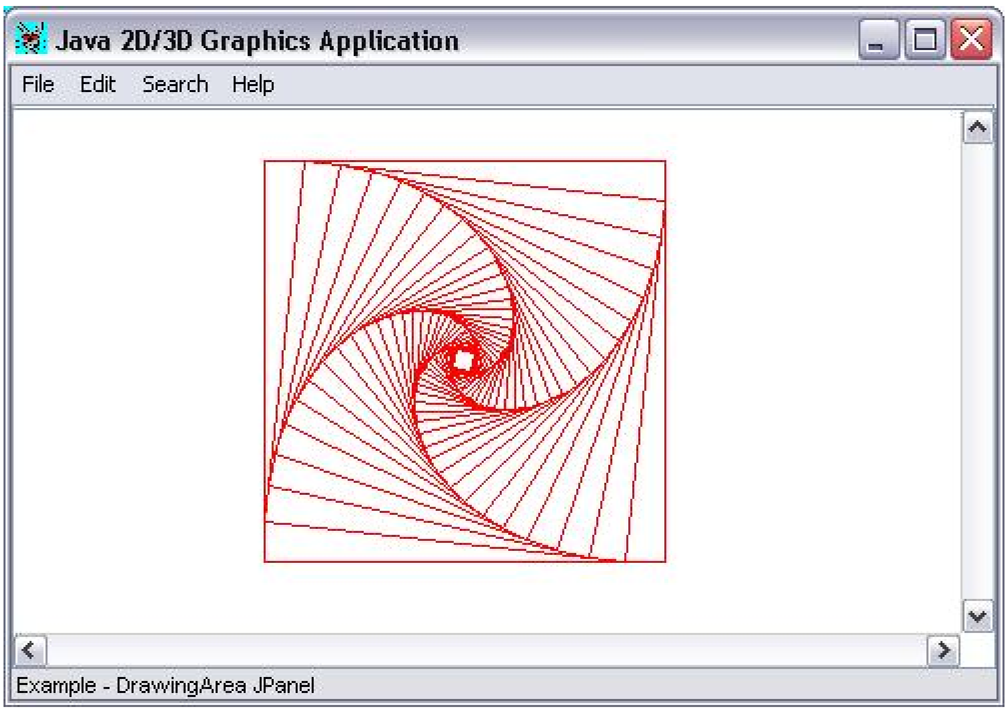


Figure 2D.4: 2D Squares wiithin a square.

Example 2D.5: Triangles within a triangle (variation).

Adapt the application developed in Exercise 3 above to draw a series of 30 triangles within a triangle to look like the one below.

Each of the triangle should be drawn offset from the previous square.

The initial triangle should have sides of 200.

Hint: If the coordinate of a vertex a (say the top center corner) is (aX, aY) the new vertex (a1X, a1Y) to create the next triangle would be as follows:

```
  a1X = (p * aX) + (q * bX)
  a1Y = (p * aY) + (q * bY)
```

Where b is the vertex for the bottom right hand corner. Try q = 0.05F & p = 1-q.

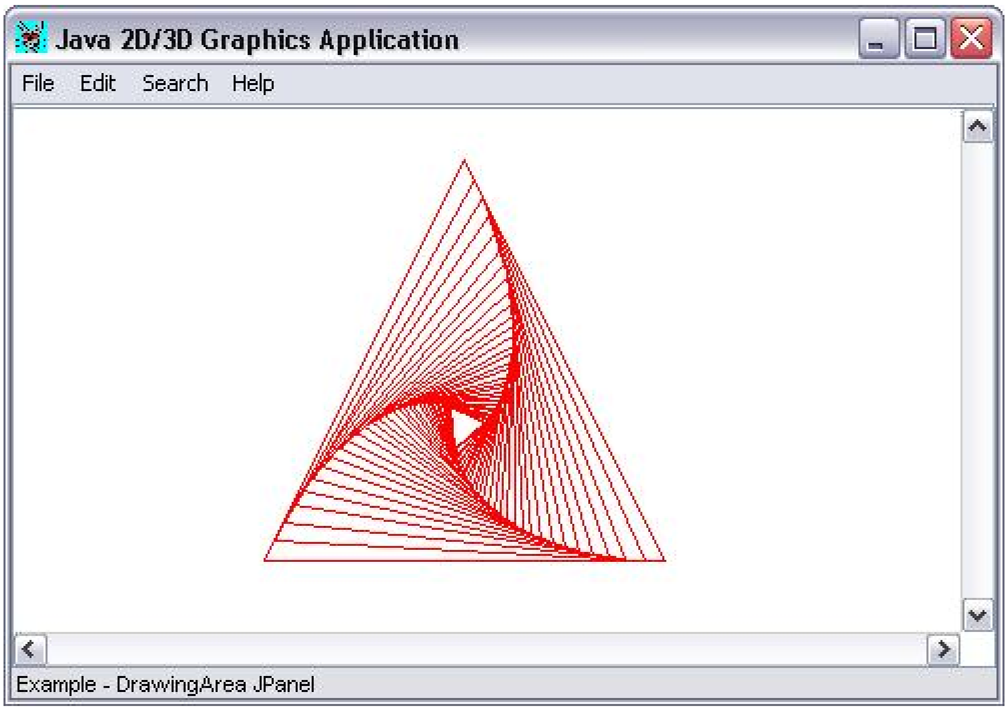


Figure 2D.5: 2D Triangles wiithin a triangle.

23 Graphics 2D: Screen Resolution

Screen resolutions can vary alot from 640x480, 800x600, 1024x768, 1280x1024 to 1600x1200. The most common being 1024 pixels in the x-axis and 768 pixels in the y-axis. The screen resolution will obviously affect, not only the quality of graphics displayed/rendered, but the size.

In the GraphicsJFrame.java application the default screen size was set and checked to make sure it was not too large. The **JFrame** was then centered as follows:

```
    setSize(500, 350);          //set default size of JFrame to width=500
height=350
    //Center the window
    Dimension screenSize = Toolkit.getDefaultToolkit().getScreenSize();
    Dimension frameSize = getSize();          //set above as 500 * 350
    if (frameSize.height > screenSize.height) //if the frame height set is
too big
    {
      frameSize.height = screenSize.height;   //set frame size to the
screen size
    }
    if (frameSize.width > screenSize.width)
    {
      frameSize.width = screenSize.width;
    }
    setLocation((screenSize.width - frameSize.width) / 2,
(screenSize.height - frameSize.height) / 2);
    setVisible(true);                         //display the JFrame
```

In some application it may be preferable to set the **JFrame** to the maximum size. This is possible by using the **setBounds** or **setSize** methods.

```
        //determine screen resolution and set to full size
        Dimension screenSize =
Toolkit.getDefaultToolkit().getScreenSize();
        setBounds(0, 0, screenSize.width, screenSize.height);
        //setSize(screenSize.width, screenSize.height); //setSize will
also work
        setVisible(true);                //display the JFrame
```

Most of the methods used here are inherited from class **java.awt.Component** i.e. **setSize**, **setLocation**, **setVisible**, **setBounds**. The class **java.awt.Dimension** uses the abstract class **Toolkit** to access the sceen size via its' methods i.e. **getDefaultToolkit**, **getScreenSize** etc.

Full-Screen Exclusive Mode API

It is possible to develop full screen applications where it is possible to suspend the windowing system (JFrame and the usual GUI components i.e. Buttons, MenuBar) so that drawing can be done directly to the screen. This has benefits in games programming where the player may feel completely immersed in the game. Additionally, a full-screen exclusive application is able to control the bit depth and size (display mode) of the screen (not be limited to the users choice).

To find out more, see the Sun Java Tutorial, Trail: Bonus; Lesson: Full-Screen Exclusive Mode API (in particular DisplayModeTest.java) and Brackeen (2003).

Example 2D.4: 2D circle within a square.

Draw a circle within a square, where the centre of the circle and square are at the centre of the screen. The side lengths of the square and the diameter of the circle should equal half the screen height. The finished program should look like the one below.

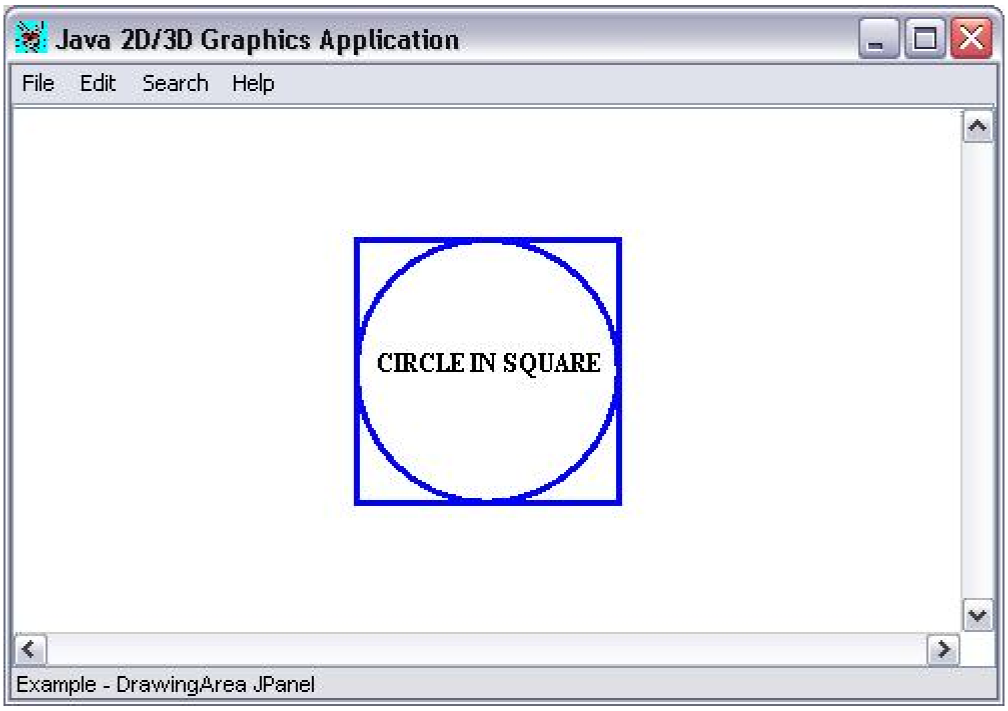


Figure 2D.4: Circle within a Square using the Dimension class.

Example 2D.5-2D: 2D Drawing grid with coloured rectangles.

Draw a grid of 10 pixels by 10 pixels onto the 'DrawingArea' making use of the Dimension class. Then place 3 coloured rectangles (180 * 90) with the top left hand corner of the green rectangle at 100, 50.

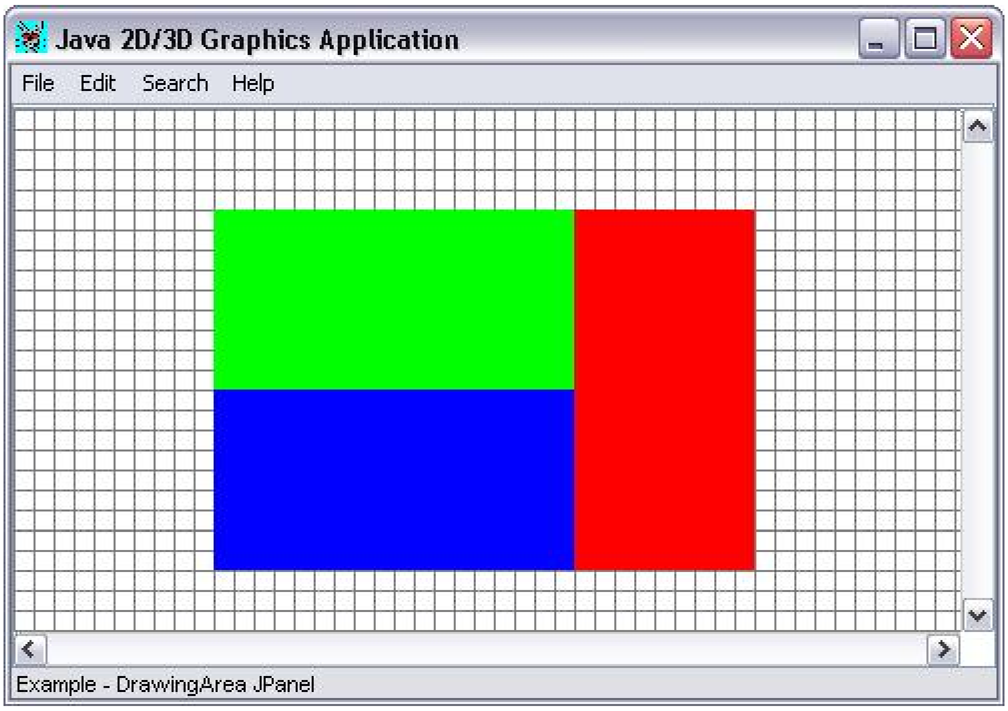


Figure 2D.5: 2D drawing grid with coloured rectangles.

24 Graphics 2D: AffineTransform

AffineTransform are briefly covered in Sun's Java Tutorial - Trail: 2DGraphics, Lesson: Displaying Graphics with Graphics2D: Transforming Shapes, Text and Images, although **AffineTransform** will also be discussed here. The **AffineTransform** class comes from the **java.awt.geom** package, so remember to to import this package at the start of your code (import java.awt.geom.*;)

AffineTransform manipulates graphics within the 'User Space' before finally rendering the tranformed graphic to the 'Device Space'.

AffineTransform offers the ability to **translate**, **rotate**, **scale**, **shear** or perform a combination (concatenate) of these manipulations on graphics.

- **translate**: specify a translation offset in the x and y directions
- **rotate**: specify an angle of rotation in radians
- **scale**: specify a scaling factor in the x and y directions
- **shear**: specify a shearing factor in the x and y directions (parallel lines remain parallel).

To use **AffineTransform** an instance must be created, then the various methods of **setToRotation**, **setToTranslation** etc. are invoked first before the **transform** method invokes the transformation, for example:

AffineTransform at = new **AffineTransform()**;//create (instantiate) a transform object (instance) at
at.**setToRotation**(-Math.PI/2.0); //invoke the setToRotation method on at
g2d.**transform(**at**)**; //invoke the transform method to finish carry-out the transformation.

AffineTransform at = new **AffineTransform()**; //create (instantiate) a transform object (instance) at
at.**setToTranslation**(25.0, 25.0); //invoke the setToTranslation method on at
at.**transform(**p,0,p,0,5**)**; //invoke the transform method to finish carry-out the transformation.

AffineTranform methods

setToTranslation , translate

setToTranslation and **translate** takes 2 arguments the translation (offset) for all x co-ordinates and the corresponding translation for all y co-ordinates.
setToTranslation(double translateX, double translateY**)**

setToRotation, rotate

setToRotation and **rotate** takes 1 or 3 arguments. The angle of rotation in radians (theta) is the first argument and the centre of rotation (the x and y co-ordinate about which rotation is

to take place) as follows:
setToRotation(double theta) or setToRotation(double theta, double x, double y)

setToScale, scale

setToScale and **scale** take arguments
setToScale(double scaleX, double scaleY)

setToShear, shear

setToShear and **shear** take arguments
setToShear(double shearX, double shearY)

setTransform, transform

setTransform and **transform** take 1 or 5 arguments. The 1 arguments, and default, would be the **AffineTransform** object itself i.e. **transform(AffineTransform** at**)**; The 5 argument method take the following:

- double array of original/source points (x,y) to be transformed
- integer of source off, the offset to the first point to be transformed in the source array, usually zero.
- double array of new/destination transformed points
- integer of destination off, the offset to the location of the first transformed point that is stored in the destination array, usually zero.
- integer number of points (x,y) in the array

Combining AffineTransform

To combine (concatenate) more than one **AffineTransform** use the non-set methods (**translate**, **rotate**, **scale**, **shear**) followed by **transform**. The reason for this is that the **AffineTransform** methods beginning with 'set' clear any previous **AffineTransform**. If only one transform is being undertaken it is safest to use 'set' (also use 'set' for the first **AffineTransform** of a series).

Beware, it is advisable, never to use setTransform to concatenate a coordinate transform onto an existing transform. The setTransform method overwrites the Graphics2D object's current transform, which might be needed for other reasons, such as positioning Swing and lightweight components in a window (See Transforming Shapes, Text and Images).

Rotation

Degrees and radians. When working with graphics objects, angles are often used. Whether for rotation of an object or the angle of a line.

Radians are used within java and it is useful to know how to covert from, the more familiar, degrees to radians.
360 degrees = 2 PI radians, therefore 180 degrees = 1 PI radians, 90 degrees = PI/2 radians etc.

(Note: Alternatively, 1 degree = 0.01745 radians. Therefore 30 degrees = 30*0.01745F. This will not be as acurate as using Math.PI/6.0)

Whilst it is possible to carry out your own conversions from degrees to radians (radians = Math.PI/180*degrees) and vice versa, java includes two conversion methods within java.awt.geom.ARC. Using a known relationship from above we could convert angles as follows:

```
  double radians = java.awt.geom.ARC.toRadians(90.0);
```

and

```
  double degrees = java.awt.geom.ARC.toDegrees(Math.PI / 2.0);
```

A clockwise rotation of 90 degress would be positive (Math.PI/2.0) and a corresponding anti-clockwise rotation would be negative (-(Math.PI/2.0)).

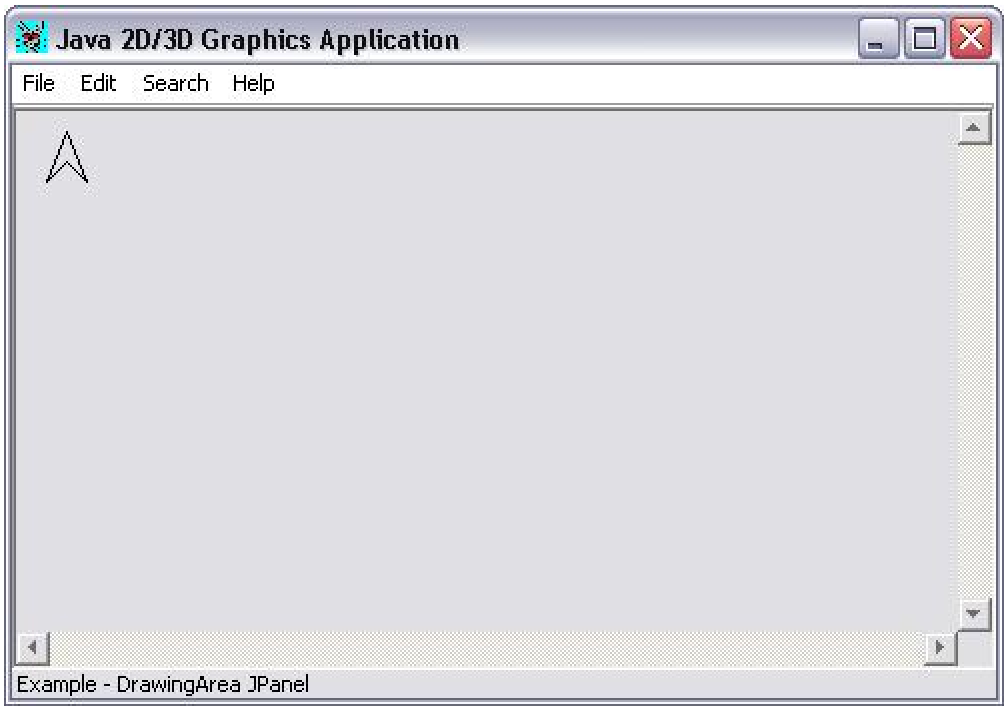
Degrees	Radians
360	2 PI
270	2PI/3.0
180	PI
90	PI/2.0
60	PI/3.0
45	PI/4.0
30	PI/6.0
1	0.01745F

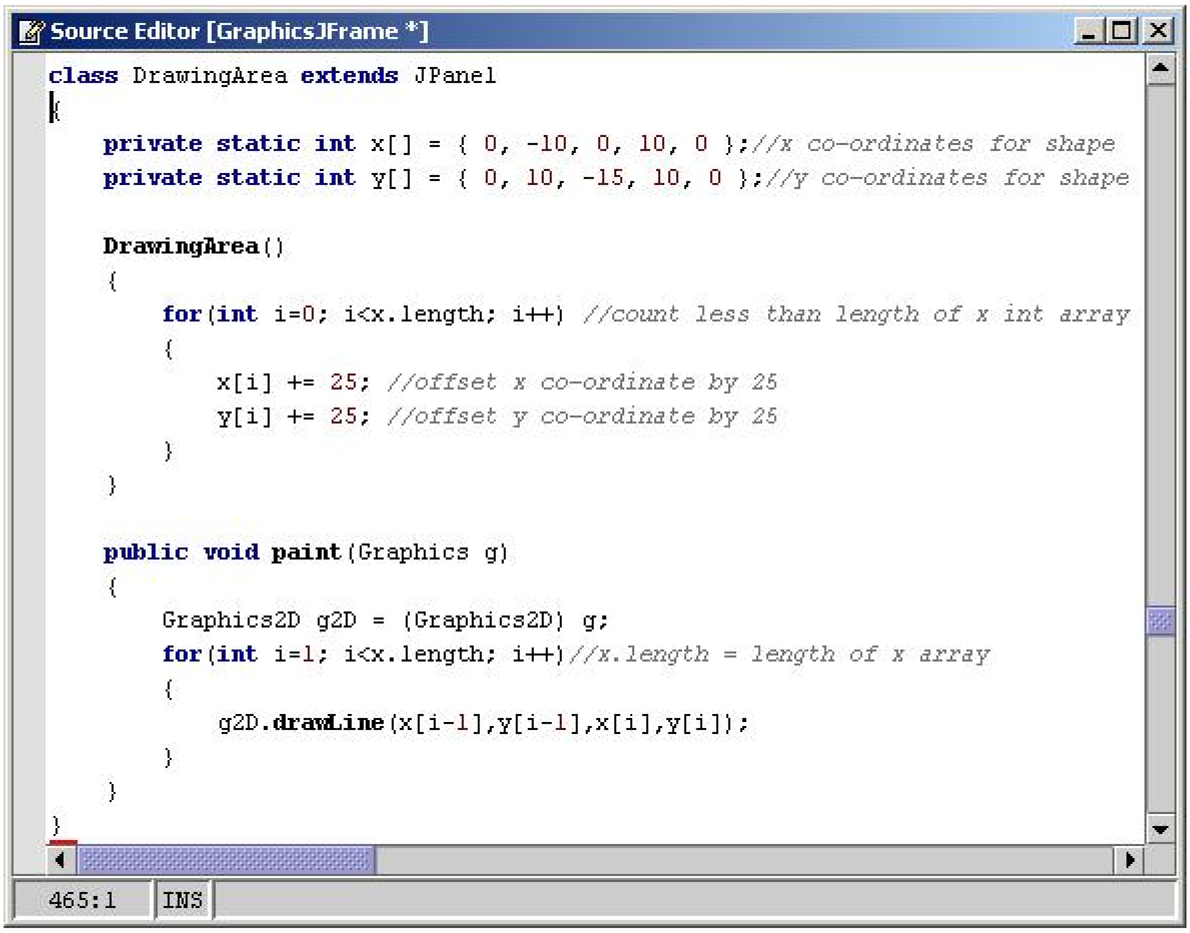
Table 2D.1: Radian to degree conversion.

Example 1: drawline used to draw shape and position on screen - simple offset

To conduct a transformation a simple shape is needed. This example creates a simple arrow-type shape, using 4 lines with the following co-ordinates starting at (0.0,0.0), (-10.0,10.0), (0.0,-15.0), (10.0,10.0) and closing at (0.0,0.0).

The shape is then translated/offset from the origin using a simple addition to its original line co-ordinates. The translation is needed because 0, 0 is the centre of the shape. A more appropriate way of producing this shape would use GeneralPath. See later.

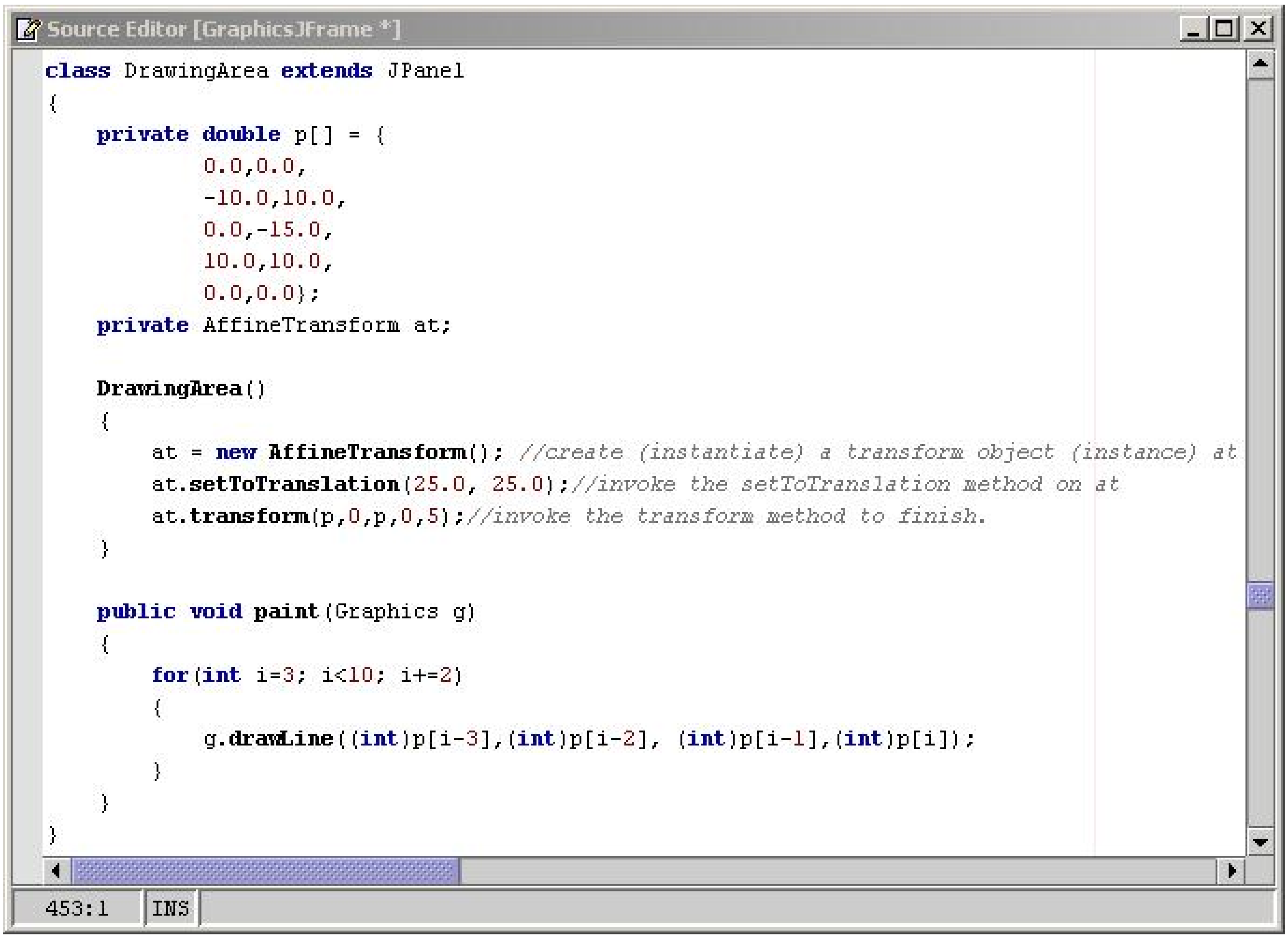




```
class DrawingArea extends JPanel
{
    private static int x[] = { 0, -10, 0, 10, 0 };//x co-ordinates for shape
    private static int y[] = { 0, 10, -15, 10, 0 };//y co-ordinates for shape

    DrawingArea()
    {
        for(int i=0; i<x.length; i++) //count less than length of x int array
        {
            x[i] += 25; //offset x co-ordinate by 25
            y[i] += 25; //offset y co-ordinate by 25
        }
    }

    public void paint(Graphics g)
    {
        Graphics2D g2D = (Graphics2D) g;
        for(int i=1; i<x.length; i++)//x.length = length of x array
        {
            g2D.drawLine(x[i-1],y[i-1],x[i],y[i]);
        }
    }
}
```

Example 2a: drawline used to draw shape and position on screen - AffineTransform offset/translation

The same drawing would be achieved using the code below, only an AffineTransform using the setToTranslataion and transform methods are used. Notice, also, that the co-ordinates are now given as a single array of type double.

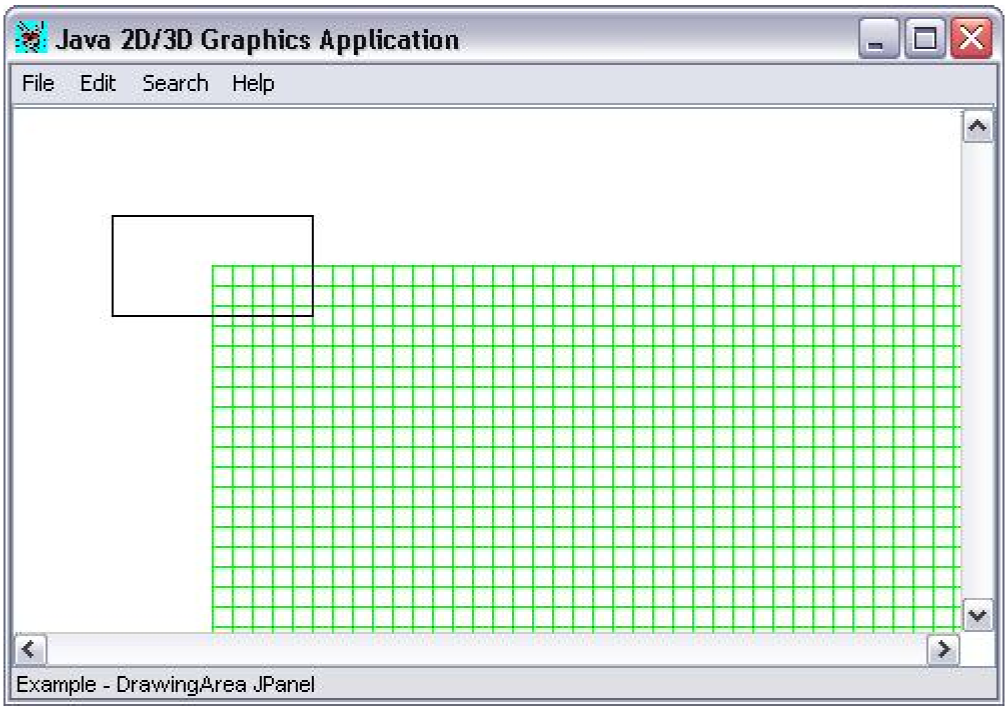

Source Editor [GraphicsJFrame *]

```
class DrawingArea extends JPanel
{
    private double p[] = {
            0.0,0.0,
            -10.0,10.0,
            0.0,-15.0,
            10.0,10.0,
            0.0,0.0};
    private AffineTransform at;

    DrawingArea()
    {
        at = new AffineTransform(); //create (instantiate) a transform object (instance) at
        at.setToTranslation(25.0, 25.0);//invoke the setToTranslation method on at
        at.transform(p,0,p,0,5);//invoke the transform method to finish.
    }

    public void paint(Graphics g)
    {
        for(int i=3; i<10; i+=2)
        {
            g.drawLine((int)p[i-3],(int)p[i-2], (int)p[i-1],(int)p[i]);
        }
    }
}
```

453:1 INS

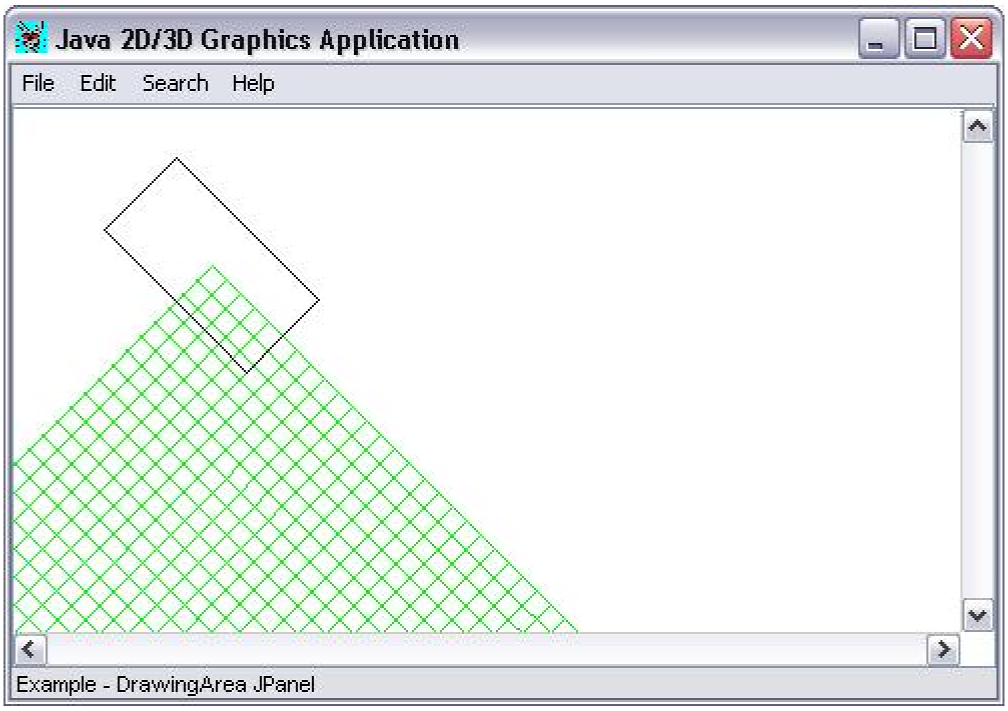
Example 2b: Rectangle2D used to draw a rectangle - AffineTransform offset/translation

By creating an AffineTransform and then using the setToTranslataion and transform methods the same type of translation can be achieved. Notice that the co-ordinates for (x, y) are given as the centroid of the shape, therefore the translation is about the centroid of the shape. This is useful in most rendering/drawing situations. The use of a grid (shown in green) illustrates that the 'user space' has been translated, not just the rectangle. The use of a grid can be very helpful when rendering and debugging transformations.



```
class DrawingArea extends JPanel
{
    AffineTransform at;
    Rectangle2D rect;

    DrawingArea()
    {
        rect = new Rectangle2D.Float(-100.0F/2, -50.0F/2, 100.0F, 50.0F);
        at = new AffineTransform();
        at.setToTranslation(100.0, 100.0);

    }
    public void paint(Graphics g)
    {
        Graphics2D g2D = (Graphics2D) g;
        g2D.setTransform(at);
        g2D.draw(rect);
    }
}
```

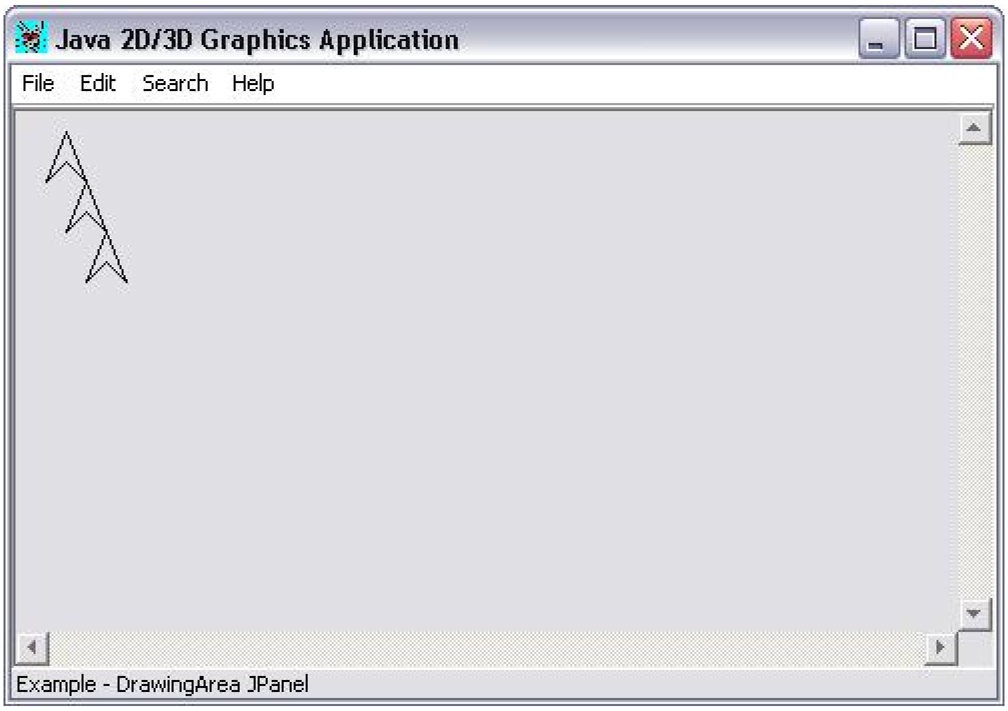
A 45 degree positive rotation would give the following:



Attempt the following exercises:

Example 3a: AffineTransform move shape down and right

Offset the initial shape as before, then draw two others shapes that are down and to the right as below.

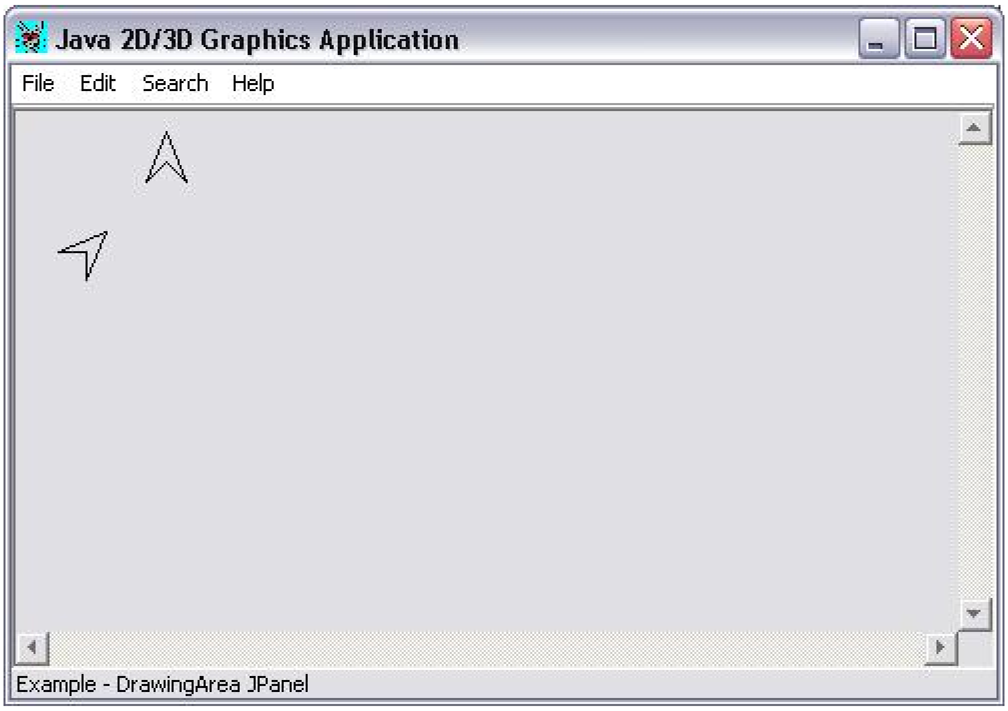


Example 3b: AffineTransform move shape down and right

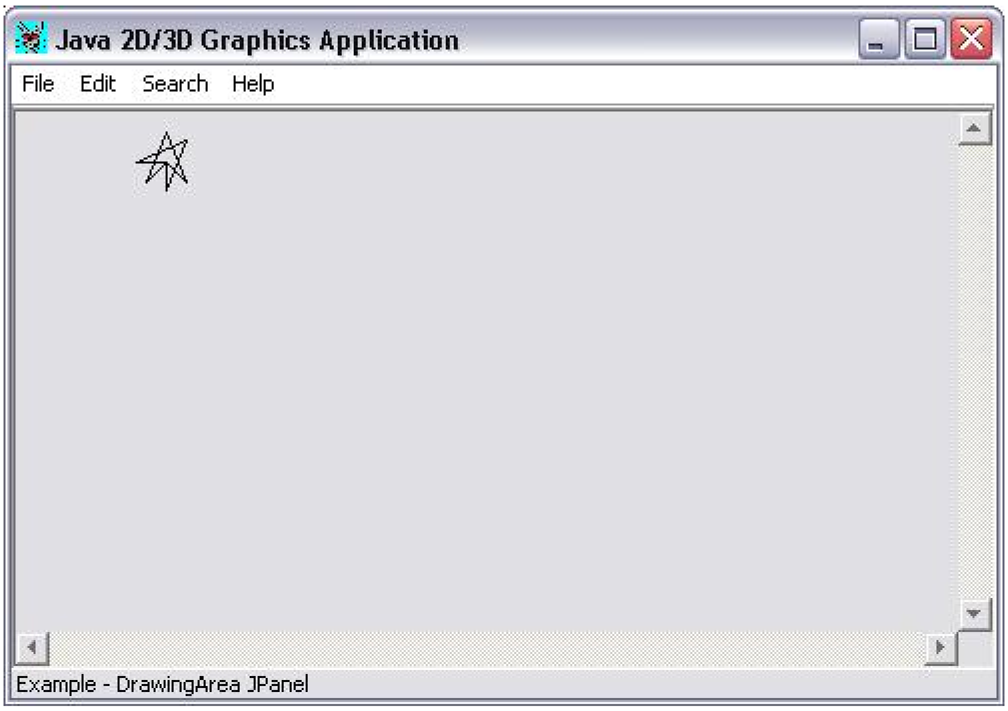
Now use the Rectangle2D from 2b.

Example 4: AffineTransform move shape down, right and rotate

Attempt to offset the initial shape as before (by 75, 25 pixels) then draw/render the shape. Next attempt to rotate the shape through 45 degrees clockwise about it axis at the same position. It is likely that the following will happen:



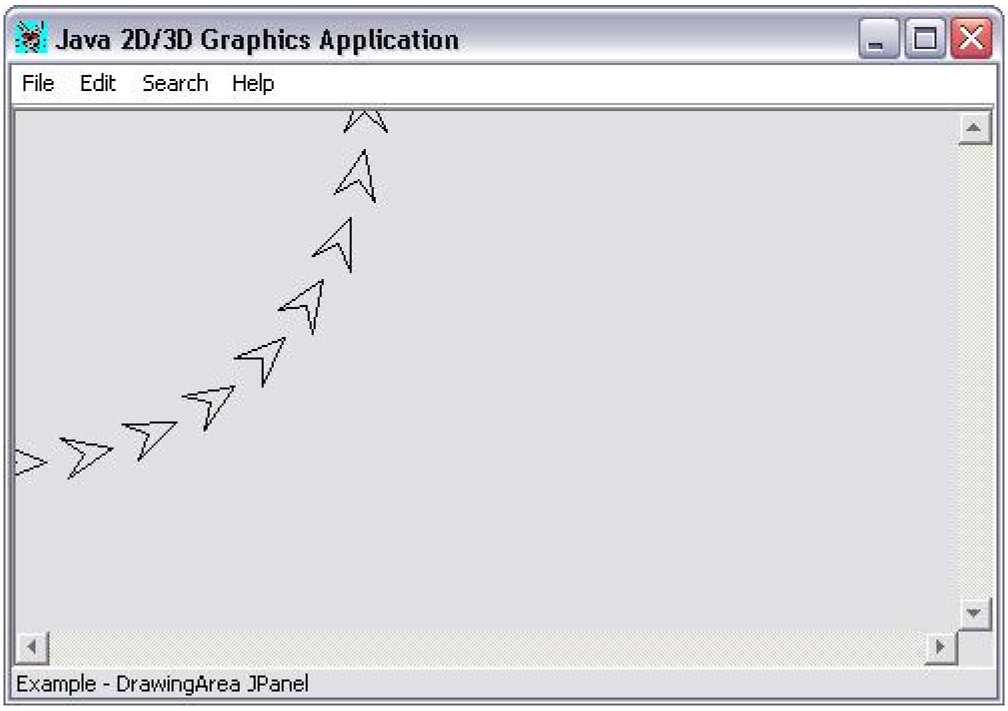
The desired outcomes is as follows:

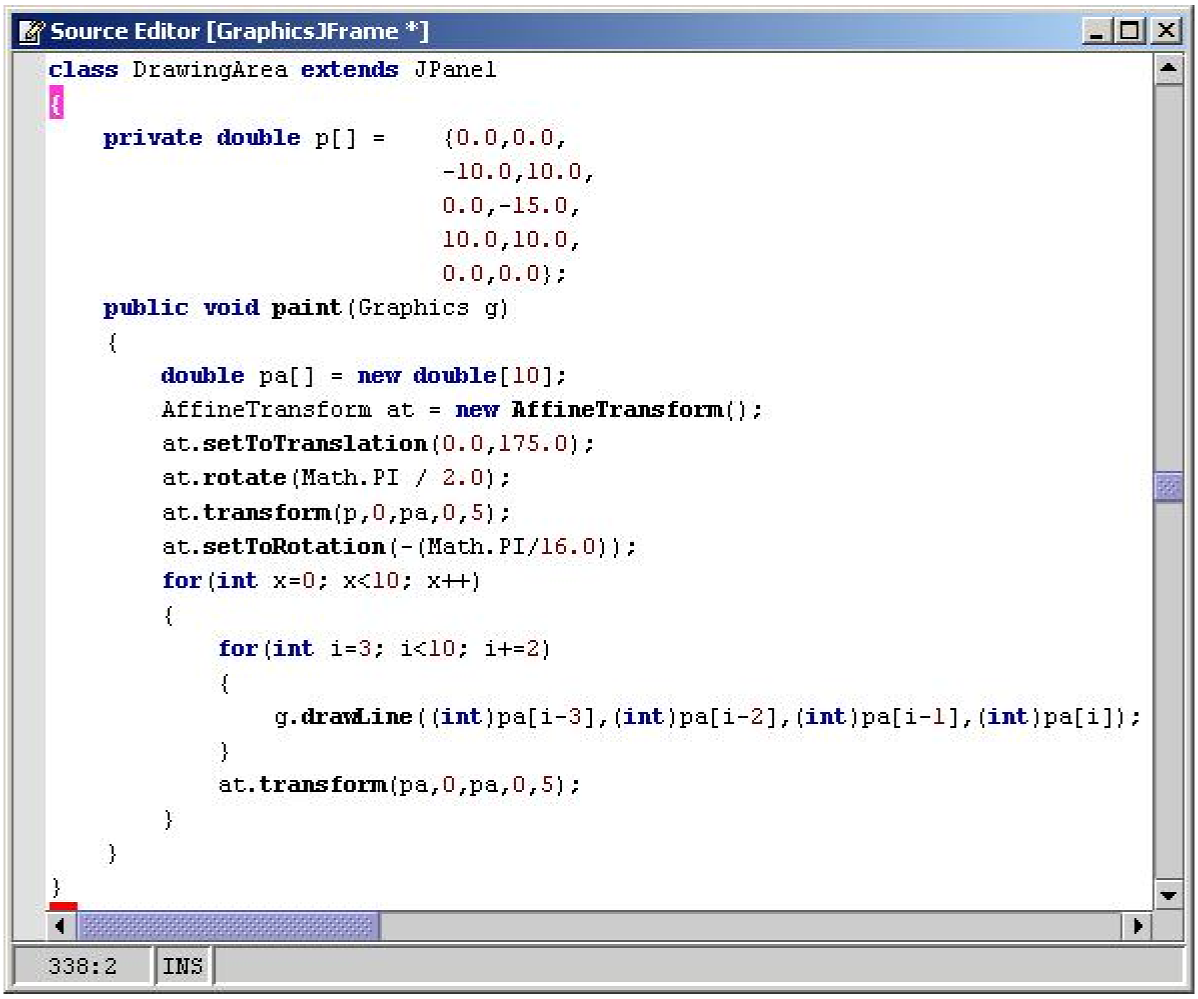


25 Graphics 2D: AffineTransform Rotation

AffineTransform Examples Continued

To rotate the shape about the origin requires an initial translate (setToTranslation) of the shape from the origin, then rotate the shape by the required amount (rotate 90 degrees or PI/2 clockwise in this case) then finally carry-out the tranformation (transform). Next the shape (in its translated and rotated form) is again rotated anticlockwise (setToRotation) drawn and transformed, drawn and transformed etc. to give the desired circular rotation. See the example below and corresponding code.



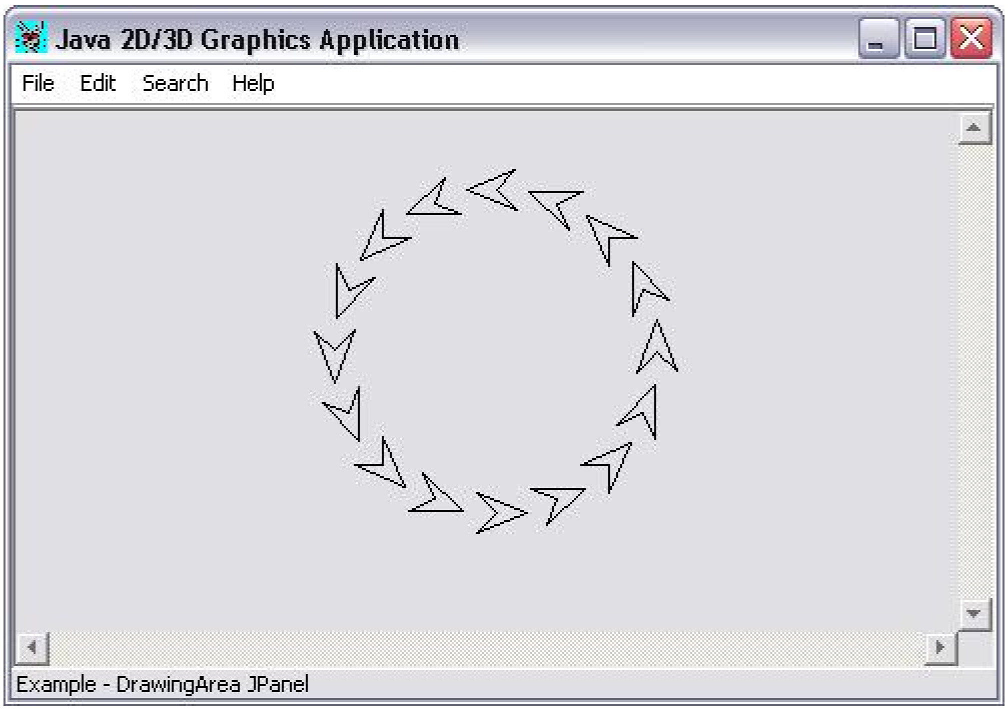


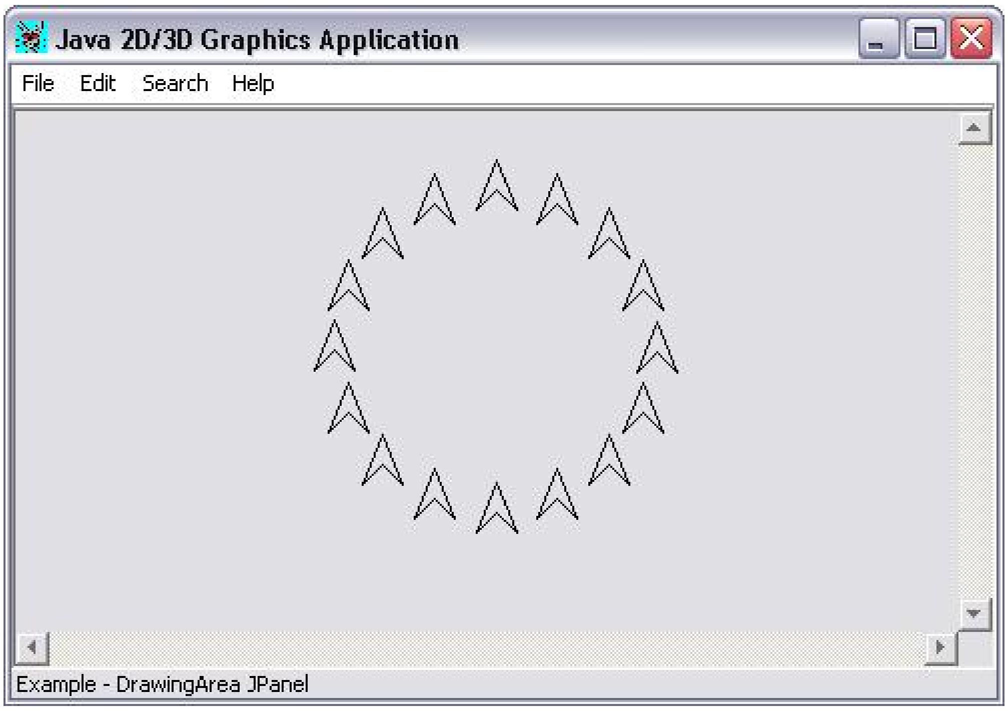
```
class DrawingArea extends JPanel
{
    private double p[] =    {0.0,0.0,
                            -10.0,10.0,
                            0.0,-15.0,
                            10.0,10.0,
                            0.0,0.0};
    public void paint(Graphics g)
    {
        double pa[] = new double[10];
        AffineTransform at = new AffineTransform();
        at.setToTranslation(0.0,175.0);
        at.rotate(Math.PI / 2.0);
        at.transform(p,0,pa,0,5);
        at.setToRotation(-(Math.PI/16.0));
        for(int x=0; x<10; x++)
        {
            for(int i=3; i<10; i+=2)
            {
                g.drawLine((int)pa[i-3],(int)pa[i-2],(int)pa[i-1],(int)pa[i]);
            }
            at.transform(pa,0,pa,0,5);
        }
    }
}
```

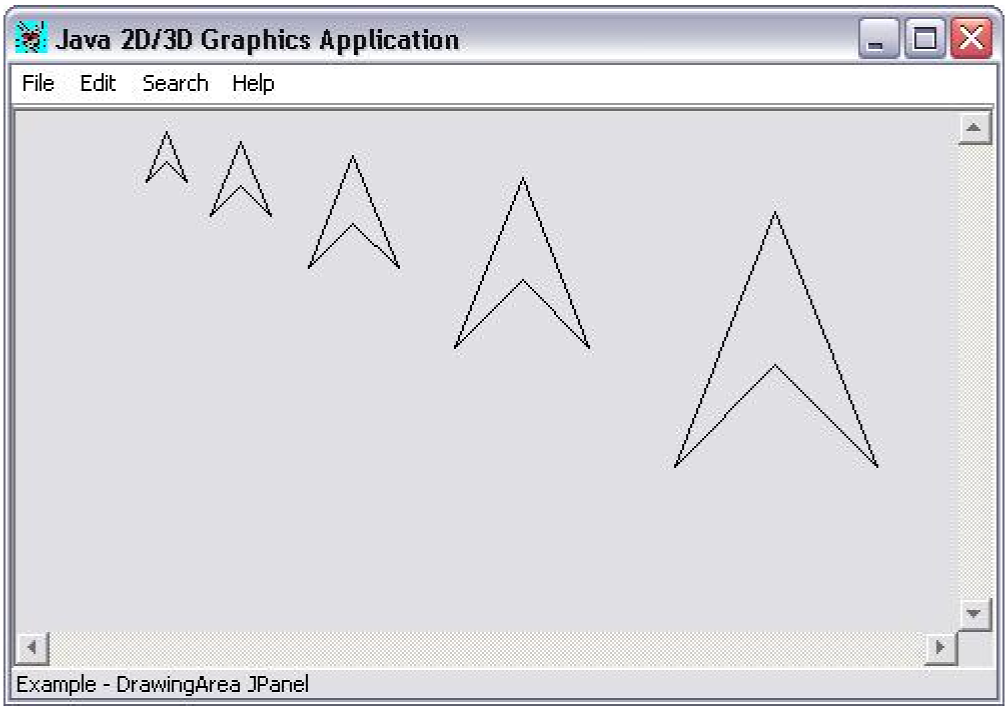
Example 5: AffineTransform rotation about any point.

Attempt to emulate the figure below, by rotating the arrow through 90 degrees as before, but then rotate about an axis other than the origin.

Hint: setToRotation and rotate take 1 or 3 arguments. The angle of rotation in radians (theta) is the first argument and the centre of rotation (the x and y co-ordinate about which rotation is to take place) as follows:
setToRotation(double theta) or setToRotation(double theta,double x, double y).


Java 2D/3D Graphics Application
File Edit Search Help
Example - DrawingArea JPanel


Java 2D/3D Graphics Application
File Edit Search Help
Example - DrawingArea JPanel


Java 2D/3D Graphics Application
File
Edit
Search
Help
Example - DrawingArea JPanel

26 Graphics 2D: GeneralPath

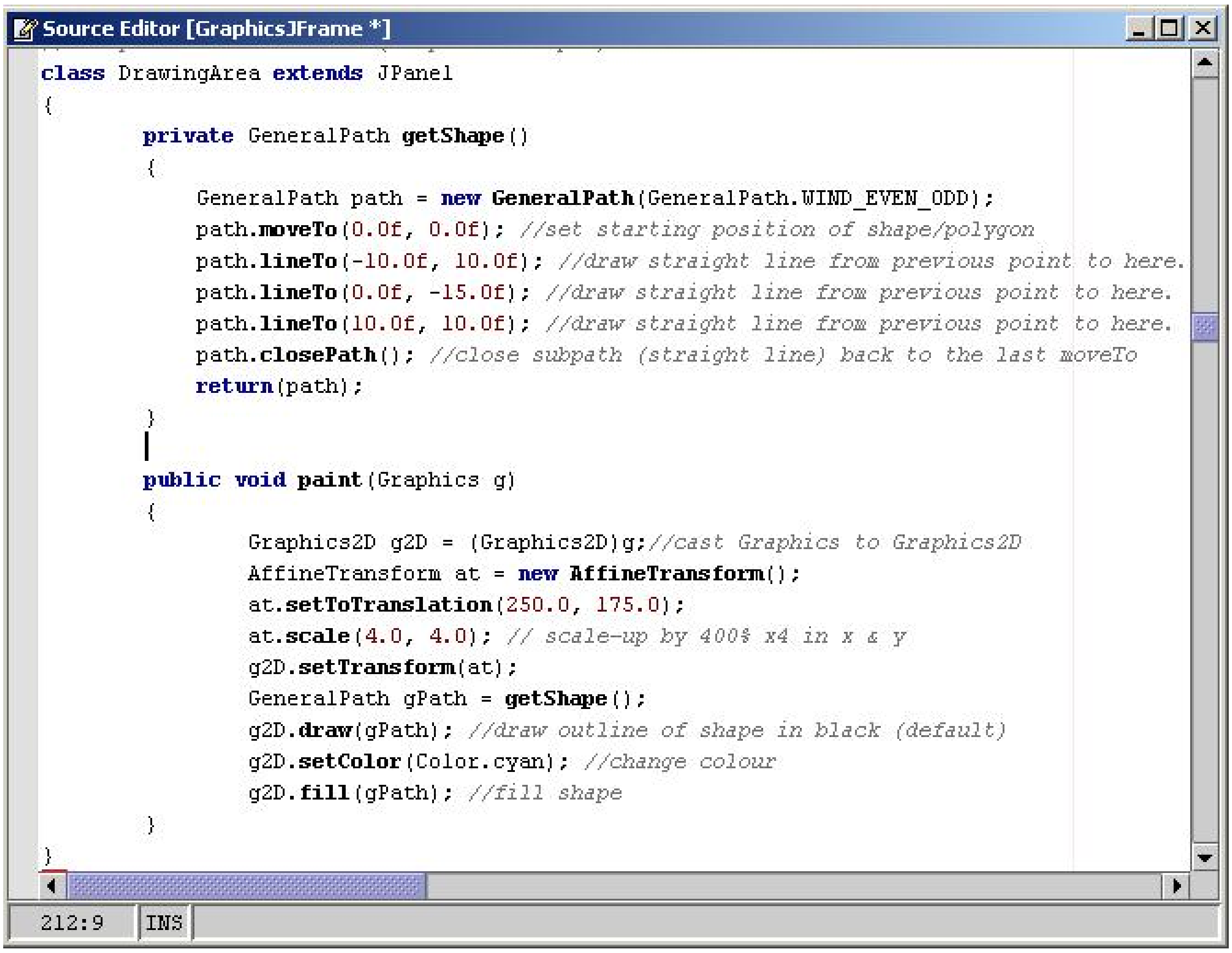
Whilst java includes a number of default shapes: rectangles; polygons; 2D lines etc. The most versitile shape/class available within the java.awt.geom package is **GeneralPath** (java.awt.geom.GeneralPath).

GeneralPath has the flexibility to enable the description of a path with any number of edges to create a complex/non-default shape. The edges can comprise of straight lines, and quadratic and cubic (Bézier) curves.

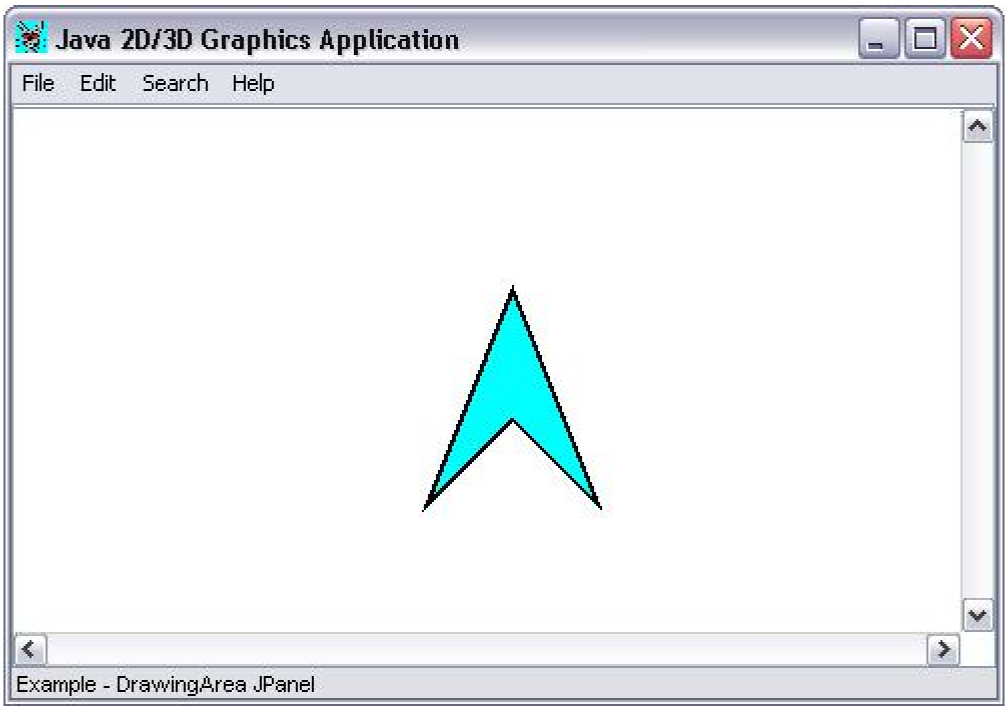
To use **GeneralPath** an instance must be created, then the various methods e.g. **moveTo**, **lineTo**, **closePath** are invoked before the object is drawn/rendered.

Example GP1: Shape using GeneralPath

To demonstrate the drawing/rendering of a shape using **GeneralPath** the arrow, previously used, is created using a new method of **GeneralPath** class called getShape().



```
class DrawingArea extends JPanel
{
        private GeneralPath getShape()
        {
            GeneralPath path = new GeneralPath(GeneralPath.WIND_EVEN_ODD);
            path.moveTo(0.0f, 0.0f); //set starting position of shape/polygon
            path.lineTo(-10.0f, 10.0f); //draw straight line from previous point to here.
            path.lineTo(0.0f, -15.0f); //draw straight line from previous point to here.
            path.lineTo(10.0f, 10.0f); //draw straight line from previous point to here.
            path.closePath(); //close subpath (straight line) back to the last moveTo
            return(path);
        }

        public void paint(Graphics g)
        {
                Graphics2D g2D = (Graphics2D)g;//cast Graphics to Graphics2D
                AffineTransform at = new AffineTransform();
                at.setToTranslation(250.0, 175.0);
                at.scale(4.0, 4.0); // scale-up by 400% x4 in x & y
                g2D.setTransform(at);
                GeneralPath gPath = getShape();
                g2D.draw(gPath); //draw outline of shape in black (default)
                g2D.setColor(Color.cyan); //change colour
                g2D.fill(gPath); //fill shape
        }
}
```



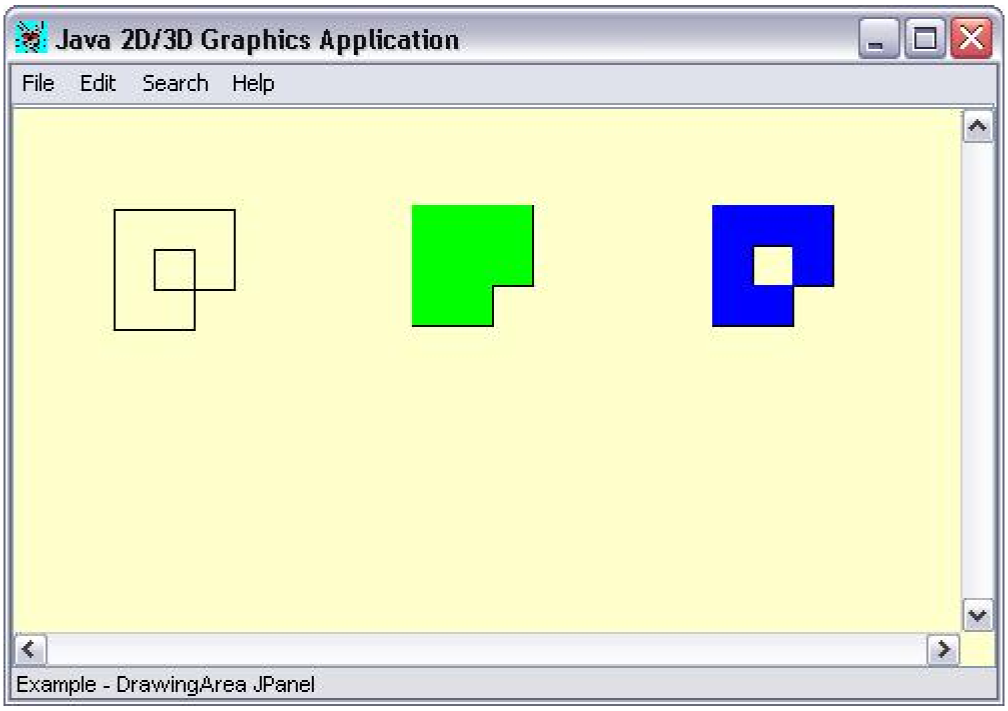
GeneralPath Winding Rules

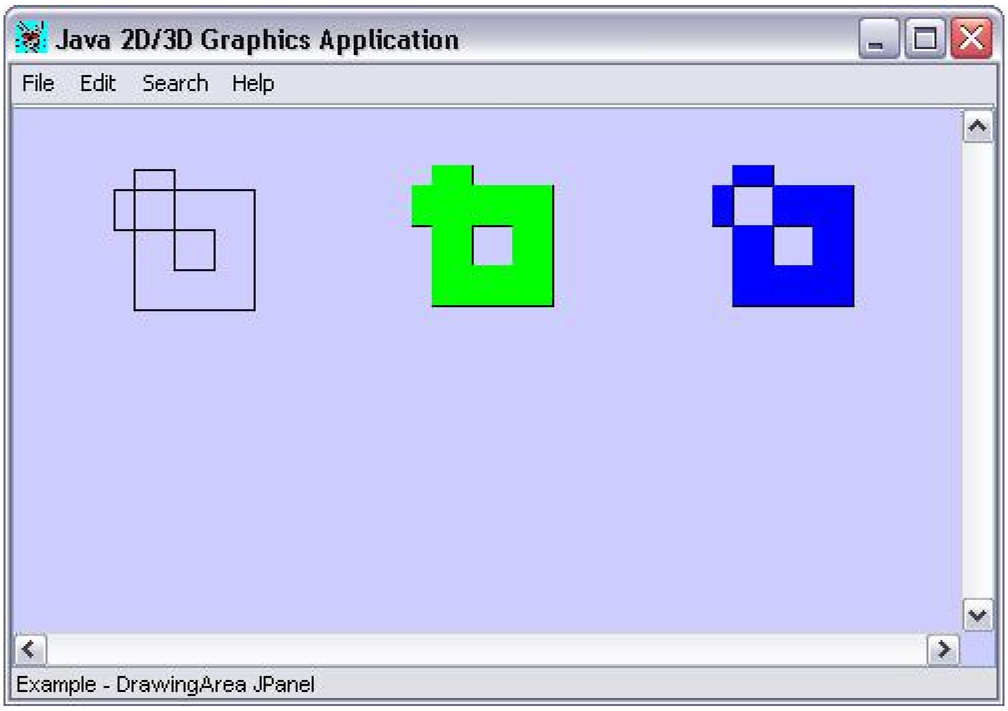
The two parameters of (static int) **GeneralPath.WIND_EVEN_ODD** or **GeneralPath.WIND_NON_ZERO** can be passed into the **GeneralPath()** constructor. These parameters represents the winding rules that tells the renderer how to determine the inside of the shape specified by the path. For standard polygons and other simple shapes the winding rules will give the same result, but with complex shapes the way that interior and exterior regions are determined can be different.

An **WIND_EVEN_ODD** winding rule means that if a drawn shape overlaps/covers a region of the shape an odd number of times the region will be filled (relates to the number of times the path lines cross). If the regions overlap an even number of times they will not be filled.

A **WIND_NON_ZERO** winding rule means that enclosed regions within a shape are filled. This is the default rule if one is not specified.

The above definitions may still seem confusing, but the two figures below show (from left to right) a shape drawn as an outline, next using the **WIND_NON_ZERO**/default rule and finally, the **WIND_EVEN_RULE**. These figures illustrate the differences in the two rules. The winding rule of the **GenerPath** can be altered using **setWindingRule** e.g. gPath.**setWindingRule**(**GeneralPath.WIND_EVEN_ODD**);


Java 2D/3D Graphics Application
File
Edit
Search
Help
Example - DrawingArea JPanel


Java 2D/3D Graphics Application
File
Edit
Search
Help
Example - DrawingArea JPanel

27 Graphics 2D: Curves

Curves are used in mathematics and computer graphics to approximate complex shapes using a finite number of mathematical points. Java 2D supports first, second and third order curves. The curves can be drawn with two end points and zero, one or two control points. zero control points are required for a straight line, one control point for a **quadratic** (second-order) and, finally, two control points for a **cubic** (third-order) Bezier curves.

When drawn/rendered the curves are pulled out toward the control point. If the control point is further away from the endpoint, the curve is pulled further toward the endpoint. Mathematically, the tangent of the curve at each endpoint is determined by a line drawn from the control point to the endpoint.

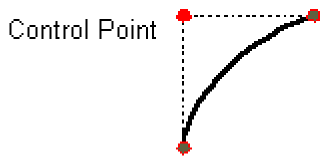


*Figure 2D.20: **quadratic** (second-order) curve (Sun)*

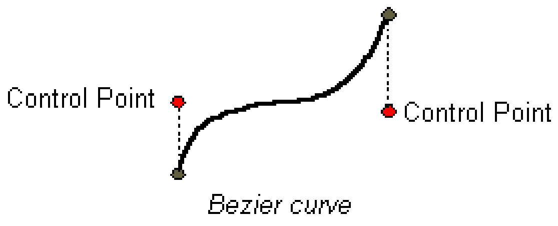


*Figure 2D.21: **cubic** (third-order) curve (Sun)*

Bezier curves are a type of parametric polynomial curve that have some very desirable properties related to computation of closed curves and surfaces.

QuadCurve2D and CubicCurve2D

This section will consider the two of classes contained within in the java.awt.geom package for drawing curves:

- **QuadCurve2D**
- **CubicCurve2D**

QuadCurve2D

The class **QuadCurve2D** and its associated subclasses **QuadCurve2D.Double**, **QuadCurve2D.Float** are used to define a quadratic parametric curve segment in (x, y) coordinate space.

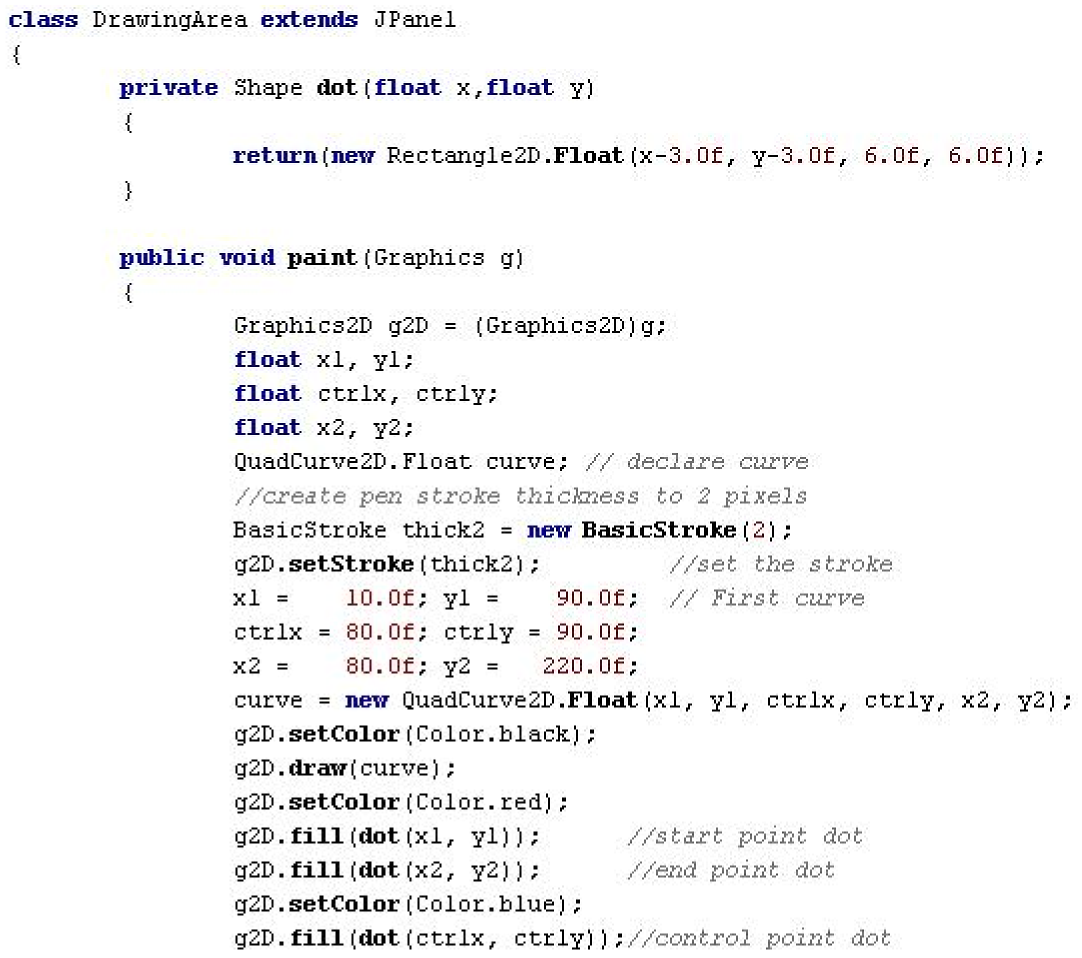
The constructors for **QuadCurve2D.Float** and **QuadCurve2D.Double** take the following form:

- **QuadCurve2D.Float(**float x1, float y1, float ctrlx, float ctrly, float x2, float y2**)**
- **QuadCurve2D.Double(**double x1, double y1, double ctrlx, double ctrly, double x2, double y2**)**

Where (x1, y1) and (x2, y2) are the two end points and (ctrlx, ctrly) is the one control points.

Example QC1: Four Quadratic Curves using QuadCurve2D

To demonstrate the drawing/rendering of quadratic curves the following snippet of code is used.



```
class DrawingArea extends JPanel
{
        private Shape dot(float x,float y)
        {
                return(new Rectangle2D.Float(x-3.0f, y-3.0f, 6.0f, 6.0f));
        }

        public void paint(Graphics g)
        {
                Graphics2D g2D = (Graphics2D)g;
                float x1, y1;
                float ctrlx, ctrly;
                float x2, y2;
                QuadCurve2D.Float curve; // declare curve
                //create pen stroke thickness to 2 pixels
                BasicStroke thick2 = new BasicStroke(2);
                g2D.setStroke(thick2);          //set the stroke
                x1 =    10.0f; y1 =    90.0f;  // First curve
                ctrlx = 80.0f; ctrly = 90.0f;
                x2 =    80.0f; y2 =   220.0f;
                curve = new QuadCurve2D.Float(x1, y1, ctrlx, ctrly, x2, y2);
                g2D.setColor(Color.black);
                g2D.draw(curve);
                g2D.setColor(Color.red);
                g2D.fill(dot(x1, y1));      //start point dot
                g2D.fill(dot(x2, y2));      //end point dot
                g2D.setColor(Color.blue);
                g2D.fill(dot(ctrlx, ctrly));//control point dot
```

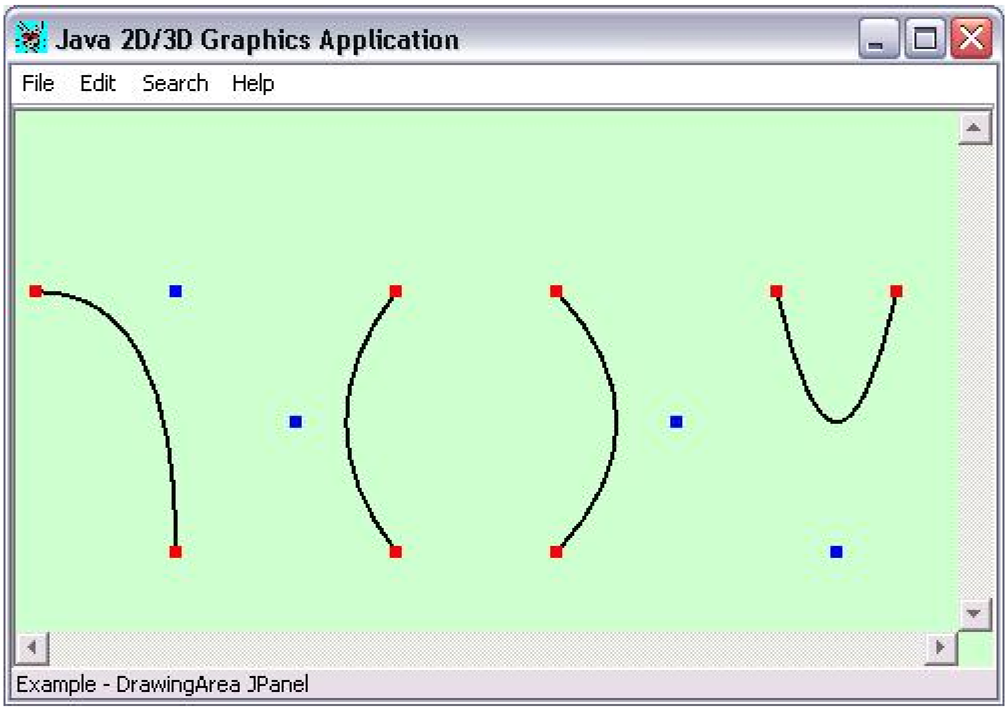
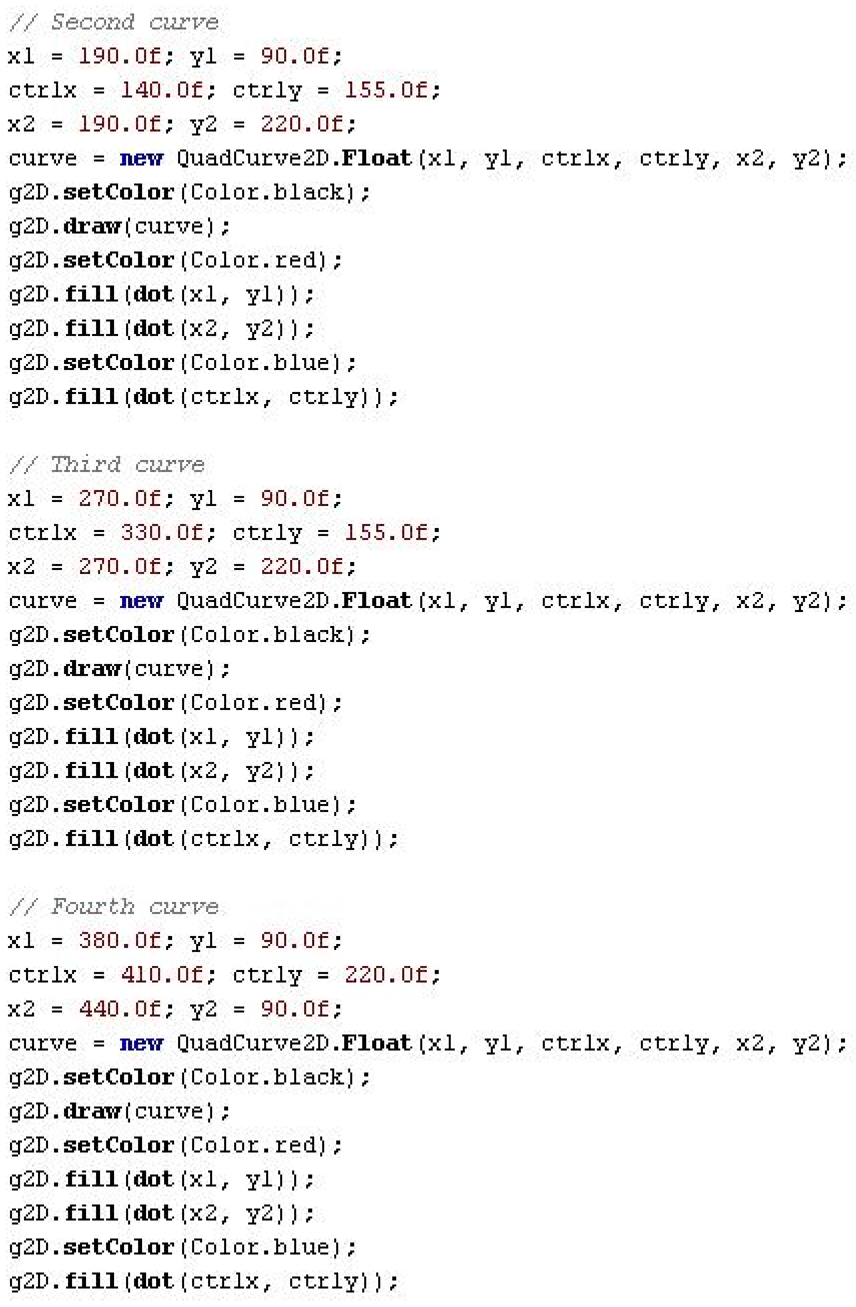


Figure 2D.22: ***QuadCurve2D*** *(second-order) curve application*

The remaining code is shown below:



```
// Second curve
x1 = 190.0f; y1 = 90.0f;
ctrlx = 140.0f; ctrly = 155.0f;
x2 = 190.0f; y2 = 220.0f;
curve = new QuadCurve2D.Float(x1, y1, ctrlx, ctrly, x2, y2);
g2D.setColor(Color.black);
g2D.draw(curve);
g2D.setColor(Color.red);
g2D.fill(dot(x1, y1));
g2D.fill(dot(x2, y2));
g2D.setColor(Color.blue);
g2D.fill(dot(ctrlx, ctrly));

// Third curve
x1 = 270.0f; y1 = 90.0f;
ctrlx = 330.0f; ctrly = 155.0f;
x2 = 270.0f; y2 = 220.0f;
curve = new QuadCurve2D.Float(x1, y1, ctrlx, ctrly, x2, y2);
g2D.setColor(Color.black);
g2D.draw(curve);
g2D.setColor(Color.red);
g2D.fill(dot(x1, y1));
g2D.fill(dot(x2, y2));
g2D.setColor(Color.blue);
g2D.fill(dot(ctrlx, ctrly));

// Fourth curve
x1 = 380.0f; y1 = 90.0f;
ctrlx = 410.0f; ctrly = 220.0f;
x2 = 440.0f; y2 = 90.0f;
curve = new QuadCurve2D.Float(x1, y1, ctrlx, ctrly, x2, y2);
g2D.setColor(Color.black);
g2D.draw(curve);
g2D.setColor(Color.red);
g2D.fill(dot(x1, y1));
g2D.fill(dot(x2, y2));
g2D.setColor(Color.blue);
g2D.fill(dot(ctrlx, ctrly));
```

CubicCurve2D

The class **CubicCurve2D** and its associated subclasses **CubicCurve2D.Double**, **CubicCurve2D.Float** are used to define a cubic parametric curve segment in (x, y) coordinate space.

The constructors for **CubicCurve2D.Float** and **CubicCurve2D.Double** take the following form:

- **CubicCurve2D.Float(**float x1, float y1, float ctrlx1, float ctrly1, float ctrlx2, float ctrly2, float x2, float y2**)**
- **CubicCurve2D.Double(**double x1, double y1, double ctrlx1, double ctrly1, double ctrlx2, double ctrly2, double x2, double y2**)**

Where (x1, y1) and (x2, y2) are the two end points and (ctrlx1, ctrly1) and (ctrlx2, ctrly2) are the two control points.

Example CC1: Four Cubic Curves using CubicCurve2D

To demonstrate the drawing/rendering of cubic curves the following snippet of code is used.

```
class DrawingArea extends JPanel
{
        public void paint(Graphics g)
        {
                Graphics2D g2D = (Graphics2D)g;
                float x1, y1;
                float ctrlx1, ctrly1;
                float ctrlx2, ctrly2;
                float x2, y2;
                CubicCurve2D.Float curve;
                //create pen stroke thickness to 2 pixels
                BasicStroke thick2 = new BasicStroke(2);
                g2D.setStroke(thick2);          //set the stroke

                // First curve
                x1 = 20.0f; y1 = 70.0f;
                ctrlx1 = 60.0f; ctrly1 = 70.0f;
                ctrlx2 = 100.0f; ctrly2 = 170.0f;
                x2 = 100.0f; y2 = 210.0f;
                curve = new CubicCurve2D.Float(
                        x1, y1, ctrlx1, ctrly1,
                        ctrlx2, ctrly2, x2, y2);
                endPtControlPt(g2D, x1, y1, ctrlx1, ctrly1);
                endPtControlPt(g2D, x2, y2, ctrlx2, ctrly2);
                g2D.draw(curve);
```

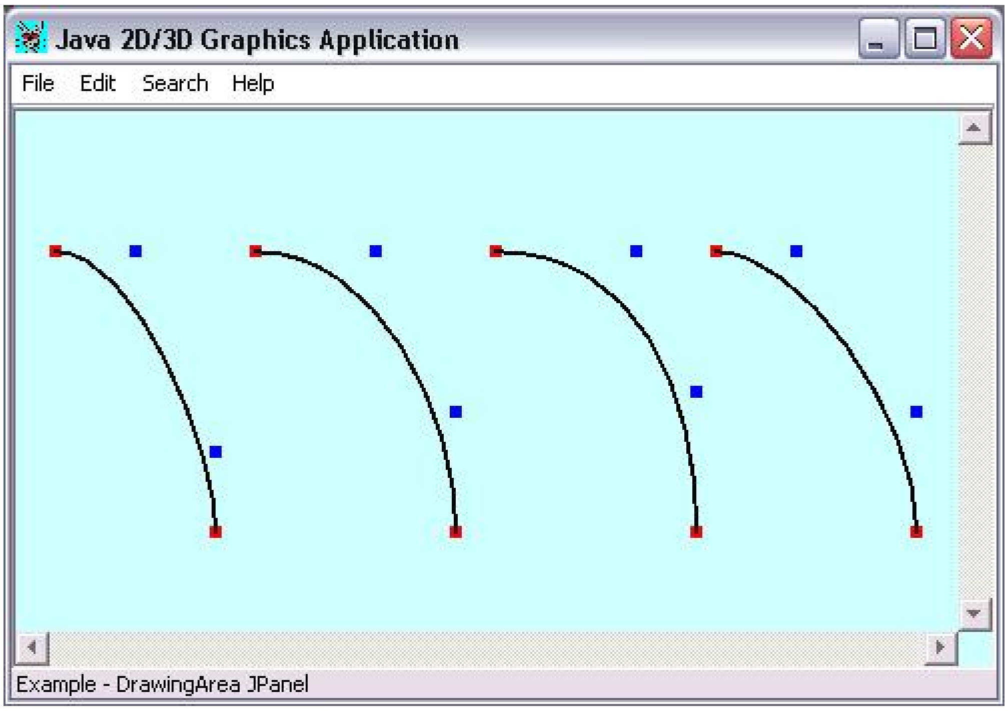
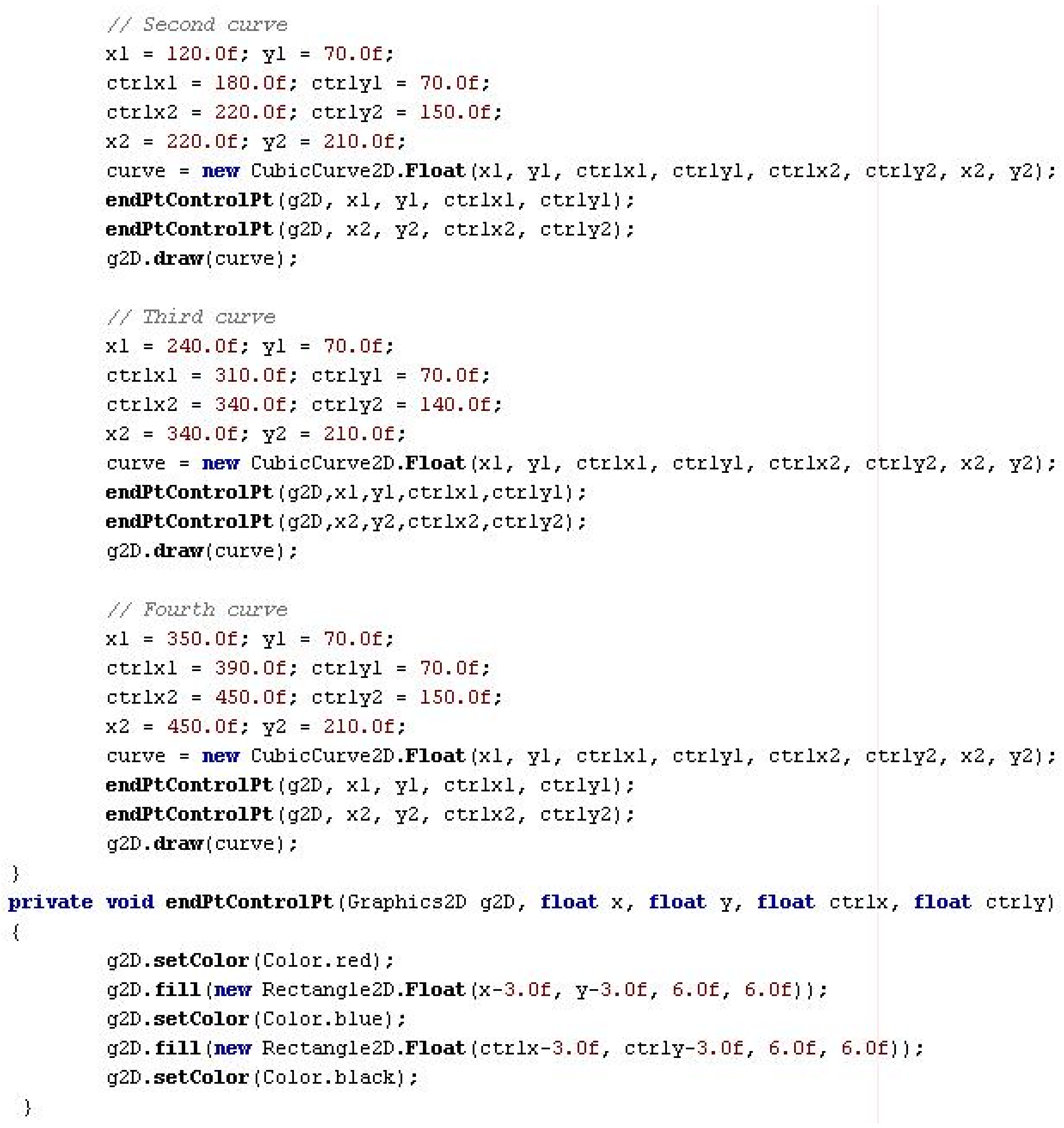


Figure 2D.23: ***CubicCurve2D*** *(third-order) curve application*

The remaining code is shown below:



```
        // Second curve
        x1 = 120.0f; y1 = 70.0f;
        ctrlx1 = 180.0f; ctrly1 = 70.0f;
        ctrlx2 = 220.0f; ctrly2 = 150.0f;
        x2 = 220.0f; y2 = 210.0f;
        curve = new CubicCurve2D.Float(x1, y1, ctrlx1, ctrly1, ctrlx2, ctrly2, x2, y2);
        endPtControlPt(g2D, x1, y1, ctrlx1, ctrly1);
        endPtControlPt(g2D, x2, y2, ctrlx2, ctrly2);
        g2D.draw(curve);

        // Third curve
        x1 = 240.0f; y1 = 70.0f;
        ctrlx1 = 310.0f; ctrly1 = 70.0f;
        ctrlx2 = 340.0f; ctrly2 = 140.0f;
        x2 = 340.0f; y2 = 210.0f;
        curve = new CubicCurve2D.Float(x1, y1, ctrlx1, ctrly1, ctrlx2, ctrly2, x2, y2);
        endPtControlPt(g2D,x1,y1,ctrlx1,ctrly1);
        endPtControlPt(g2D,x2,y2,ctrlx2,ctrly2);
        g2D.draw(curve);

        // Fourth curve
        x1 = 350.0f; y1 = 70.0f;
        ctrlx1 = 390.0f; ctrly1 = 70.0f;
        ctrlx2 = 450.0f; ctrly2 = 150.0f;
        x2 = 450.0f; y2 = 210.0f;
        curve = new CubicCurve2D.Float(x1, y1, ctrlx1, ctrly1, ctrlx2, ctrly2, x2, y2);
        endPtControlPt(g2D, x1, y1, ctrlx1, ctrly1);
        endPtControlPt(g2D, x2, y2, ctrlx2, ctrly2);
        g2D.draw(curve);
}
private void endPtControlPt(Graphics2D g2D, float x, float y, float ctrlx, float ctrly)
{
        g2D.setColor(Color.red);
        g2D.fill(new Rectangle2D.Float(x-3.0f, y-3.0f, 6.0f, 6.0f));
        g2D.setColor(Color.blue);
        g2D.fill(new Rectangle2D.Float(ctrlx-3.0f, ctrly-3.0f, 6.0f, 6.0f));
        g2D.setColor(Color.black);
 }
```

28 Graphics 2D: General Path Curves

This section will consider the two of methods available within in the **GeneralPath** class for drawing curves:

- **quadTo()**
- **curveTo()**

Previously the **.moveTo();**, **.lineTo();** and **.closePath();** methods of **GeneralPath** were introduced:

- **moveTo()** for start position of a path (start coordinates required)
- **lineTo()** for straight line (end coordinates required)
- **closePath()** close the subpath (no coordinates required as it closes the path to the previous moveTo coordinates)

Now for curves:

- **quadTo()** for second-order quadratic curve (one control coordinate and end coordinates)
- **curveTo()** for third-ordered cubic Bezier curve (two control coordinates and end coordinates)

The methods **quadTo()** and **curveTo()** take the following form:

- **quadTo(**float ctrlx, float ctrly, float x2, float y2**)**

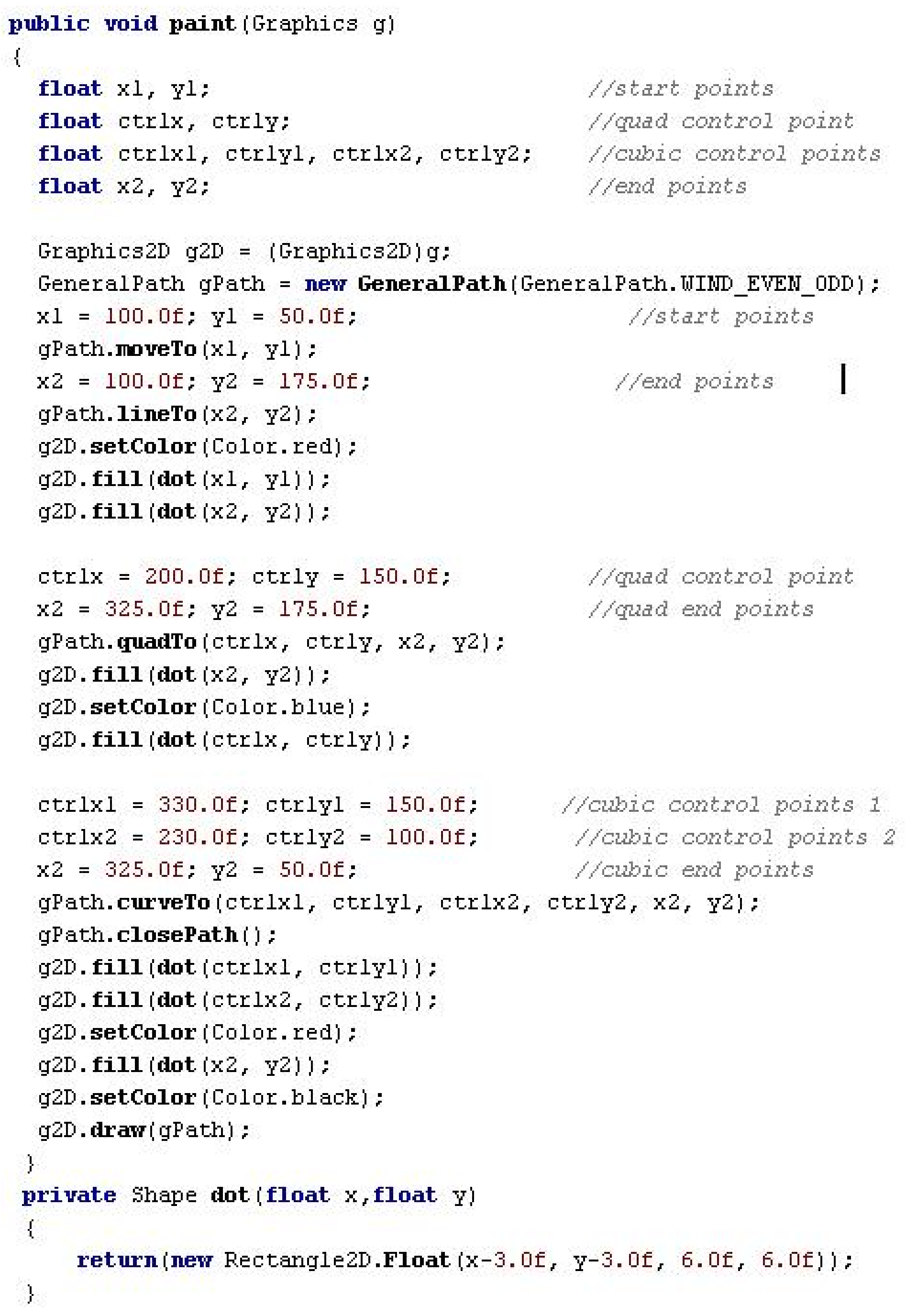
quadTo adds a curved segment, defined by two new points, to the path by drawing a Quadratic (second order) curve that intersects
both the current coordinates and the coordinates (x2, y2), using the specified point (ctrlx, ctrly) as a quadratic parametric control point.

- **curveTo(**float ctrlx1, float ctrly1, float ctrlx2, float ctrly2, float x2, float y2**)**

curveTo adds a curved segment, defined by three new points, to the path by drawing a Cubic (Bézier / third order) curve that intersects both the current coordinates and the coordinates (x2, y2), using the specified points (ctrlx1, ctrly1) and (ctrlx2, ctrly2) as Cubic / Bézier control points.

Example QTCT1: quadTo and cubeTo Curves

To demonstrate the drawing/rendering of quadratic and cubic curves the following snippet of code is used.



```
public void paint(Graphics g)
{
  float x1, y1;                              //start points
  float ctrlx, ctrly;                        //quad control point
  float ctrlx1, ctrly1, ctrlx2, ctrly2;      //cubic control points
  float x2, y2;                              //end points

  Graphics2D g2D = (Graphics2D)g;
  GeneralPath gPath = new GeneralPath(GeneralPath.WIND_EVEN_ODD);
  x1 = 100.0f; y1 = 50.0f;                      //start points
  gPath.moveTo(x1, y1);
  x2 = 100.0f; y2 = 175.0f;                    //end points
  gPath.lineTo(x2, y2);
  g2D.setColor(Color.red);
  g2D.fill(dot(x1, y1));
  g2D.fill(dot(x2, y2));

  ctrlx = 200.0f; ctrly = 150.0f;            //quad control point
  x2 = 325.0f; y2 = 175.0f;                  //quad end points
  gPath.quadTo(ctrlx, ctrly, x2, y2);
  g2D.fill(dot(x2, y2));
  g2D.setColor(Color.blue);
  g2D.fill(dot(ctrlx, ctrly));

  ctrlx1 = 330.0f; ctrly1 = 150.0f;        //cubic control points 1
  ctrlx2 = 230.0f; ctrly2 = 100.0f;         //cubic control points 2
  x2 = 325.0f; y2 = 50.0f;                  //cubic end points
  gPath.curveTo(ctrlx1, ctrly1, ctrlx2, ctrly2, x2, y2);
  gPath.closePath();
  g2D.fill(dot(ctrlx1, ctrly1));
  g2D.fill(dot(ctrlx2, ctrly2));
  g2D.setColor(Color.red);
  g2D.fill(dot(x2, y2));
  g2D.setColor(Color.black);
  g2D.draw(gPath);
 }
 private Shape dot(float x,float y)
 {
     return(new Rectangle2D.Float(x-3.0f, y-3.0f, 6.0f, 6.0f));
 }
```

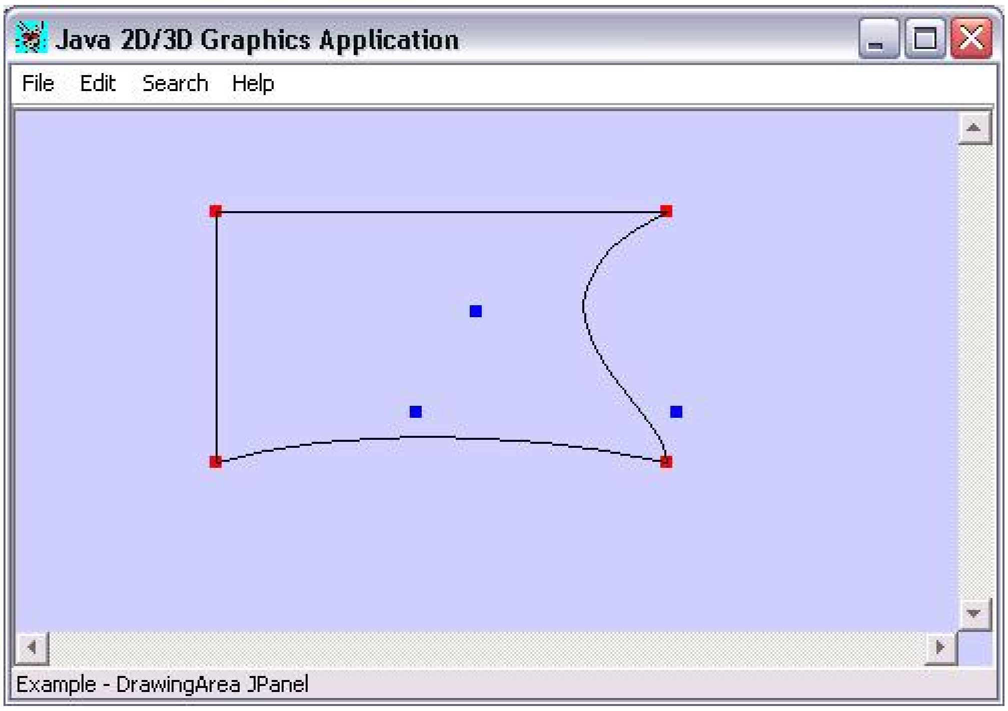


*Figure 2D.24: **quadTo** (second-order) & **cubeTo** (third-order) curve application*

The code above is verbose, to assist with the understanding of the **quadTo** and **cubeTo** methods. Concise code demonstrating the above shape can be seen below:

```
class DrawingArea extends JPanel
{
  public void paint(Graphics g)
  {
    Graphics2D g2d = (Graphics2D)g;
    GeneralPath gPath = new GeneralPath(GeneralPath.WIND_EVEN_ODD);
    gPath.moveTo(0.0f, 0.0f);
    gPath.lineTo(0.0f, 125.0f);
    gPath.quadTo(100.0f, 100.0f, 225.0f, 125.0f);
    gPath.curveTo(230.0f, 100.0f, 130.0f, 50.0f, 225.0f, 0.0f);
    gPath.closePath();

    AffineTransform at = new AffineTransform();
    at.setToTranslation(100.0f, 50.0f);
    g2d.transform(at);
    g2d.setColor(Color.blue);
    g2d.fill(gPath);
  }
}
```

29 Graphics 2D: Constructive Area Geometry

Constructive Area Geometry (CAG) is briefly covered in Sun's Java Tutorial - Trail: 2DGraphics, Lesson: Displaying Graphics with Graphics2D: Constructing Complex Shapes from Geometry Primitives, although CAG will briefly be discussed here.

Constructive area geometry (CAG) enables the creation of complex geometric shapes by performing boolean operations on standard shapes (e.g. rectangles, ellipses, and polygons). The **Shape** that supports boolean operations is **Area** from the **java.awt.geom** package.

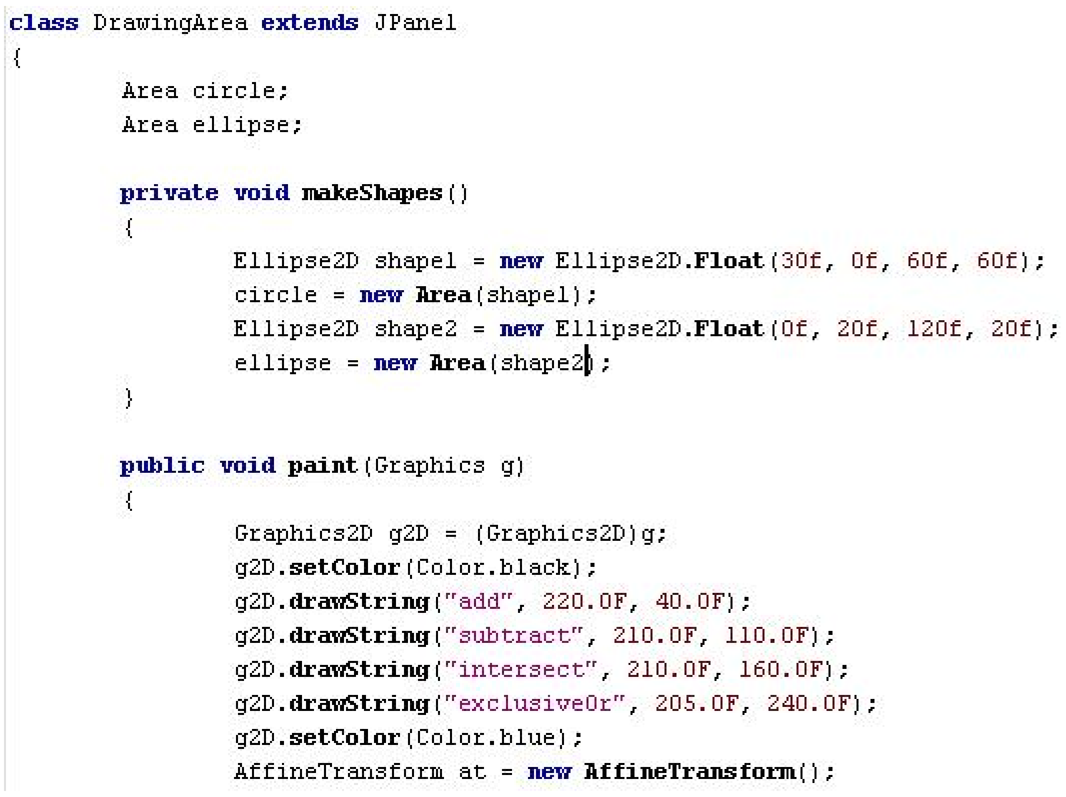
The **Area** class enables the CAG operations of **add** (union), **subtract**, **intersect** and **exclusiveOr**. First the shape objects need to be created, then the **Area** constructor is used to create an **Area** instance for each shape i.e:

```
Ellipse2D shape1 = new Ellipse2D.Float(30f, 0f, 60f, 60f);
circle = new Area(shape1); // Area circle previously declared

Ellipse2D shape2 = new Ellipse2D.Float(0f, 20f, 120f, 20f);
ellipse = new Area(shape2); // Area ellipse previously declared
```

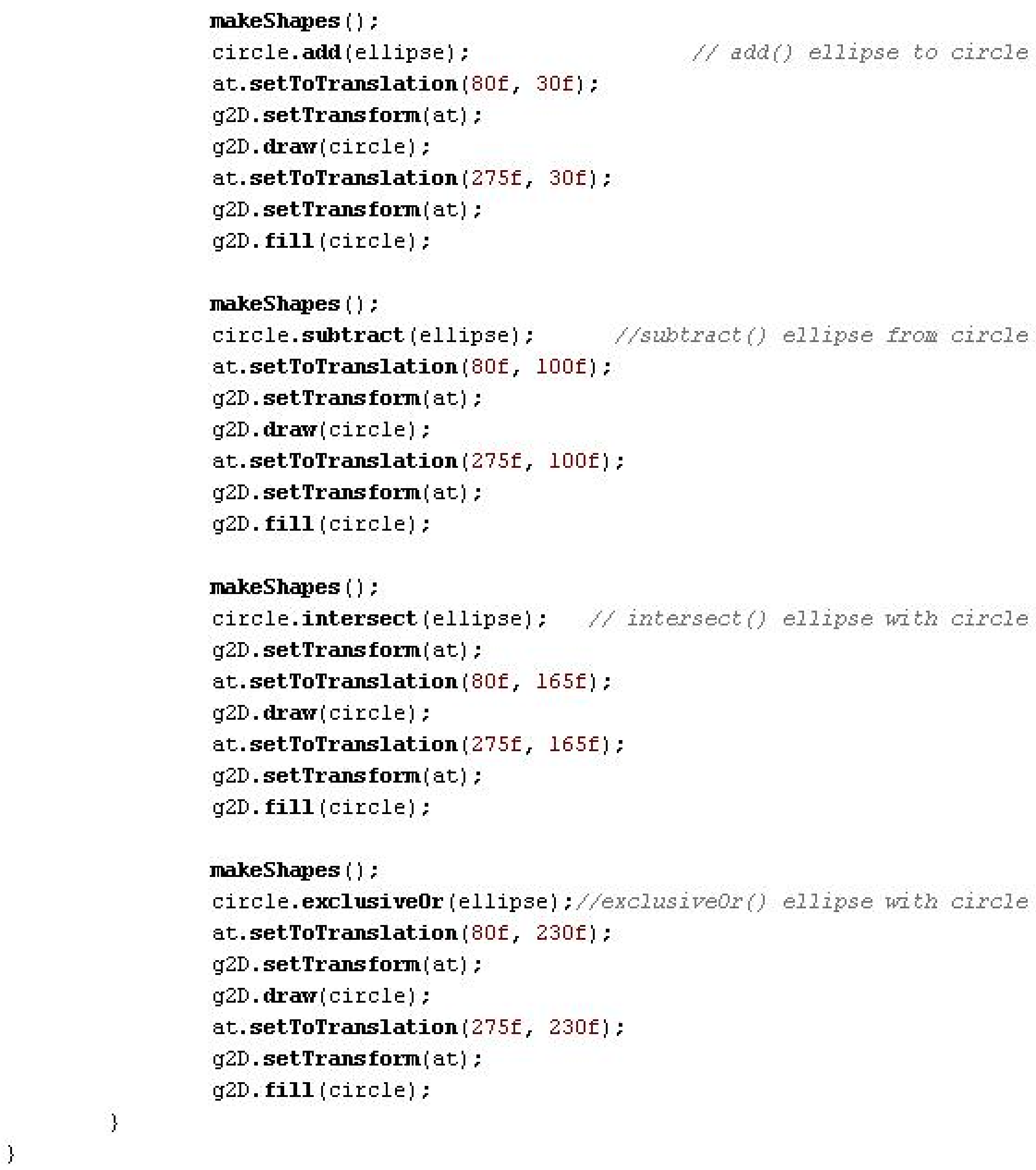
The concept of CAG is best illustrated by the following example.

Example CAG1: add, subtract, intersect and exclusiveOr: ellipse with circle

To demonstrate the drawing/rendering of CAG shapes the following code is used.



```
class DrawingArea extends JPanel
{
        Area circle;
        Area ellipse;

        private void makeShapes()
        {
                Ellipse2D shape1 = new Ellipse2D.Float(30f, 0f, 60f, 60f);
                circle = new Area(shape1);
                Ellipse2D shape2 = new Ellipse2D.Float(0f, 20f, 120f, 20f);
                ellipse = new Area(shape2);
        }

        public void paint(Graphics g)
        {
                Graphics2D g2D = (Graphics2D)g;
                g2D.setColor(Color.black);
                g2D.drawString("add", 220.0F, 40.0F);
                g2D.drawString("subtract", 210.0F, 110.0F);
                g2D.drawString("intersect", 210.0F, 160.0F);
                g2D.drawString("exclusiveOr", 205.0F, 240.0F);
                g2D.setColor(Color.blue);
                AffineTransform at = new AffineTransform();
```



```
                makeShapes();
                circle.add(ellipse);                // add() ellipse to circle
                at.setToTranslation(80f, 30f);
                g2D.setTransform(at);
                g2D.draw(circle);
                at.setToTranslation(275f, 30f);
                g2D.setTransform(at);
                g2D.fill(circle);

                makeShapes();
                circle.subtract(ellipse);      //subtract() ellipse from circle
                at.setToTranslation(80f, 100f);
                g2D.setTransform(at);
                g2D.draw(circle);
                at.setToTranslation(275f, 100f);
                g2D.setTransform(at);
                g2D.fill(circle);

                makeShapes();
                circle.intersect(ellipse);   // intersect() ellipse with circle
                g2D.setTransform(at);
                at.setToTranslation(80f, 165f);
                g2D.draw(circle);
                at.setToTranslation(275f, 165f);
                g2D.setTransform(at);
                g2D.fill(circle);

                makeShapes();
                circle.exclusiveOr(ellipse);//exclusiveOr() ellipse with circle
                at.setToTranslation(80f, 230f);
                g2D.setTransform(at);
                g2D.draw(circle);
                at.setToTranslation(275f, 230f);
                g2D.setTransform(at);
                g2D.fill(circle);
        }
}
```

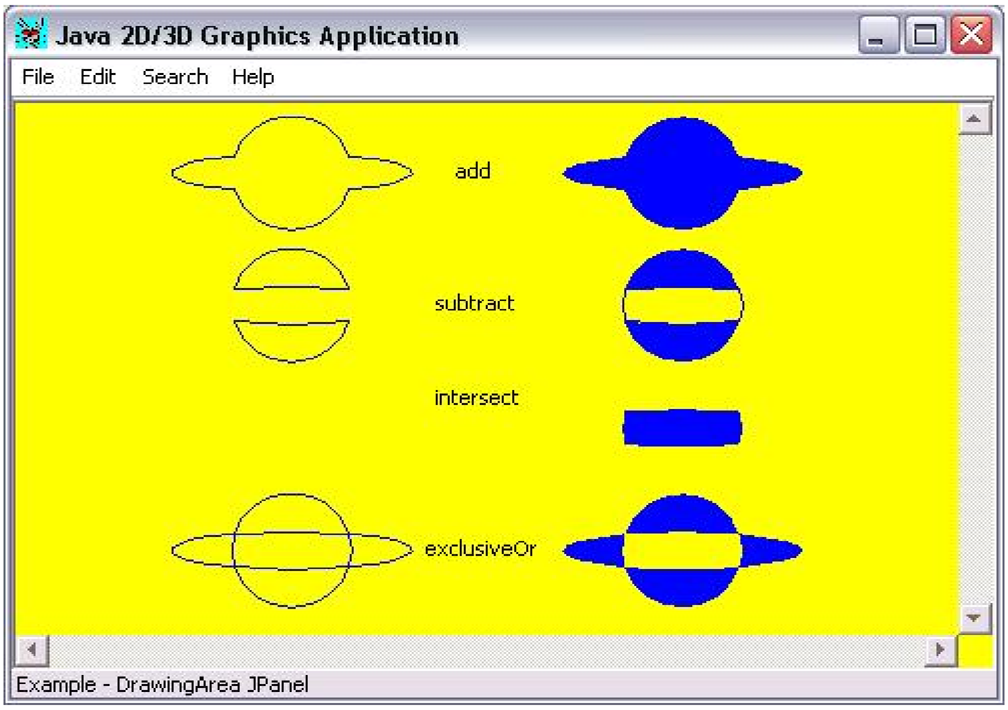


Figure 2D.25: Constructive Area Geometry application (add, subtract, intersect & exclusiveOr)