

Special Topics No. 5: Vector Sample Program

The Vector class is introduced in Chapter 9. In this document, we will implement a class called `PlottingCanvas` that allows the programmers to plot a graph by passing two-dimensional (x,y) points to a `PlottingCanvas`. A `PlottingCanvas` object draws the graph by drawing a line between every successive pair of points. As we described in Special Topics No. 4: Nondisappearing Shapes, for the drawn graphics to remain visible, we must redraw them in the paint method. For the `PlottingCanvas` class, we must remember the points on the graph so we can redraw the lines between every pair of adjacent points. We will use a vector to keep track of these points.

Introduction

In this document, we will implement a sample class called `PlottingCanvas` that uses a vector to keep track of the points in a graph. In the Special Topics document No. 4, we described the technique of putting all code pertaining to the drawing inside the paint method and inside other methods that are called from the paint method to make the drawn objects remain visible on the window. We will use this technique in the `PlottingCanvas` class.

In addition to describing the use of a vector for the class, we will explain the logical-to-pixel coordinate mapping and other drawing-related topics. Since we want to focus on the use of vectors, the `PlottingCanvas` class will have only very simple drawing functionality.

1 Using the PlottingCanvas Class

Before we get into the internal workings of the `PlottingCanvas` class, we will first present how to use the class. To draw a graph on a `PlottingCanvas` object, we must place it on a `Frame` object (actually, any `Container` object will do, but eventually we need a top-level `Frame` object to make the whole thing appear on the screen). We can use a `MainWindow` or an `Applet`, for example. In the following examples, we will use a `MainWindow` object. Here's a short program that draws a diamond by passing five points to a `PlottingCanvas` object `canvas` using its `setNextPoint` method.

```
import javabook.*;

class TestCanvas
{
    public static void main( String[] args )
    {
        MainWindow    myWindow = new MainWindow
                        ( "Testing the Plotting Window..." );
        PlottingCanvas canvas   = new PlottingCanvas( );

        myWindow.setSize(800, 600);
        myWindow.add( canvas );
        myWindow.setVisible( true );

        canvas.setUnit(2);

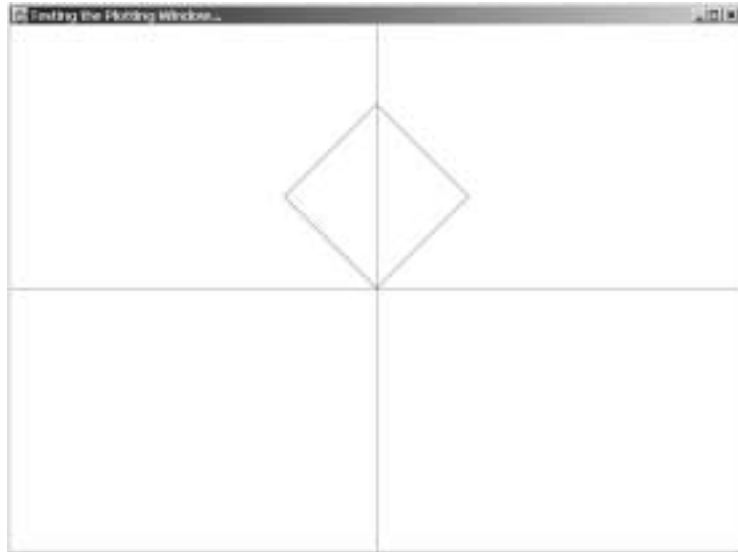
        canvas.setNextPoint( 0.0, 0.0);
        canvas.setNextPoint( 50.0, 50.0);
        canvas.setNextPoint( 0.0, 100.0);
        canvas.setNextPoint(-50.0, 50.0);
        canvas.setNextPoint( 0.0, 0.0);
    }
}
```

This command places **canvas** on **myWindow** so it becomes visible when **myWindow** is made visible..

Pass the points in the graph to draw.

This sets each logical unit to cover 2 pixels on the window.

Running the TestCanvas program will result in the following window:



We add the PlottingCanvas object *canvas* to *myWindow* as

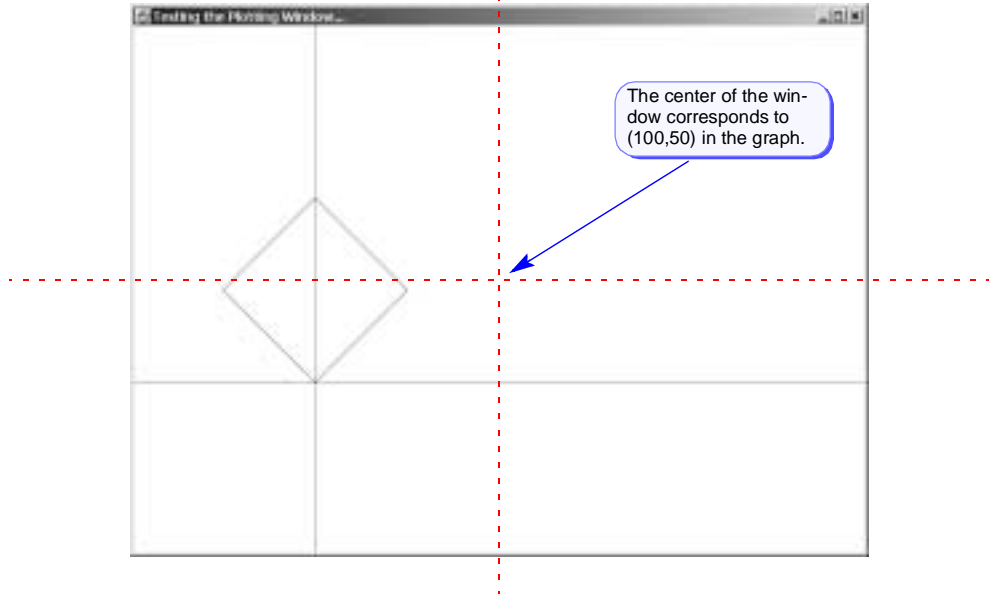
```
myWindow.add( canvas );
```

We discussed this `add` method in Chapter 5. This makes the canvas visible when its container *myWindow* appears on the screen. Please refer to Chapter 5 for more discussion on how the layout manager affects the placement of the added components. We do not have to worry about the layout here because we are placing only one object.

There are two methods to set up the properties of a *PlottingCanvas* object. The first method is `setUnit`. By using this method, we can set the number of pixels to use for each logical unit. In this sample program, we are setting 2 pixels per logical unit, so if there are 50 logical units between the two points, then there will be 100 pixels between the two points on the screen, for example. This is the way to enlarge or shrink the graph. Pass larger values to enlarge and smaller values to shrink. We can view the value we pass to the `setUnit` method as the magnification factor. For example, passing the value of 2 means double magnification, passing 0.5 means one-half magnification; which is half reduction. Try different values and see how the graph changes.

The second method is `setOrigin`. We did not use this method in the sample program, so the *canvas* made the center of the window the origin point of the graph. We can use the `setOrigin` method to change the location of the origin

point on the window. Suppose we want to shift the origin point 100 units to the left and 50 units down to make the graph appear as the following:



We can do so by calling the `setOrigin` method as

```
canvas.setOrigin( 100, 50 );
```

This will result in making the center of the window correspond to the point (100, 50) in the graph, consequently, the origin of the graph will be at 100 units left and 50 units down from the center of the window. Try different values and see how the origin point shifts on the window.

To draw a graph, we pass the (x,y) coordinate points in the graph in order, i.e., we pass the first point in the graph, then the second point, then the third point, and so forth. A `PlottingCanvas` will connect the passed points to draw a graph. In the sample program, we passed five points that correspond to the end-points of a diamond using the `setNextPoint` method. A more general use is to compute (x, y) coordinates as a result of some computation. For example, the following code will plot a simple linear function:

```
double y;
```

```

for (int x = 0; x < 300; x += 5 ) {

    y = 2 * x + 5;

    canvas.setNextPoint( x, y );
}

```

For a lot more interesting graphs, run the following TestCanvas2 program.

```

import javabook.*;

class TestCanvas2
{
    public static void main( String[] args )
    {
        MainWindow      myWindow = new MainWindow
                        ( "Testing the Plotting Window..." );
        PlottingCanvas canvas    = new PlottingCanvas( );

        //myWindow.setSize(800, 600);
        myWindow.add( canvas );
        myWindow.setVisible( true );

        //canvas.setOrigin(100, 50);
        canvas.setUnit(2);

        double centerX = 0.0;
        double centerY = 0.0;
        double radian, x, y, r;

        r = 150; //radius; if you don't change the value
                //of r inside the for loop, it will
                //draw a circle

        for (int angle = 0; angle <= 360; angle += 2) {

            radian = Math.toRadians( angle );

            //try different formula for computing r
            //change the constants such as 200, 5, and 10
            //and see what happens.

            r = 200 * Math.sin(radian)
              * Math.cos(radian) * Math.cos(radian);
            // r = 200 * Math.sin( 5 * radian );
            // r = 10 * radian;

```

```

        x = centerX + r * Math.cos( radian );
        y = centerY + r * Math.sin( radian );

        canvas.setNextPoint( x, y );
    }
}

```

2 Defining the PlottingCanvas Class

Before continuing with the implementation of the `PlottingCanvas` class, you should be familiar with the use of the `paint` and `repaint` methods. If you are not, please read Special Topics No. 4 before continuing this section. Because we need to use the `paint` method, we need to define the class as a subclass of some component. We will define the class as a subclass of `Canvas`, that provides a template for generic components. (Note: If you want to create a Swing version of `PlottingCanvas`, then you should define the class as subclass of `JPanel`.)

Since it is very difficult to predict the number of points in the graph to draw, using a vector to remember the points makes sense. Instead of using a default initial size and increment amount, we will set them ourselves. The declaration and corresponding initialization in the constructor are as follows:

```

private Vector pt;

...
public PlottingCanvas( )
{
    ...
    pt = new Vector( 100, 25 );
    ...
}

```

Every time the `setNextPoint` is called, we create a new point object with the given `x` and `y` values and add this new point to the `pt` vector. We could use the `Point2D.Double` class in the `java.awt.geom` package instead of the `Point` class in the `java.awt` package because we want to use double values for higher precision. The `Point` class only allows `int` values for `x` and `y`. Next, the `repaint` method is called to refresh the graph so the newly added point will be included in the drawing. Finally, the counter data member `currentPt` is incremented by one. Here's the method:

```

public void setNextPoint( double x, double y )

```

```

{
    pt.add( new Point2D.Double( x, y ) );

    repaint( );

    currentPt++;
}

```

Notice that we call `repaint` before incrementing the counter `currentPt`, because the value of `currentPt` before the increment is the index of the last point in `pt` due to the zero-based indexing of a vector.

Calling the `repaint` method will eventually cause the `paint` method to be executed. The `paint` method of the `PlottingCanvas` class will draw the axes and the lines between the consecutive pairs of points in the `pt` vector. Here's the method:

```

public void paint(Graphics g)
{
    drawAxis(g);
    plotLines(g);
}

```

Before we go over the `drawAxis` and `plotLines` private methods, we need to explain how the logical-to-pixel coordinate mapping works. We will be using one of the drawing method `drawLine` of the `Graphics` class to draw a line between two points. (Note: The drawing methods are introduced in Chapter 2.) Drawings in the `Graphics` class use the pixel coordinates, while the points we pass in the `setNextPoint` method use the logical coordinates. So we need a way to map the values in the logical coordinates to those in the pixel coordinates.

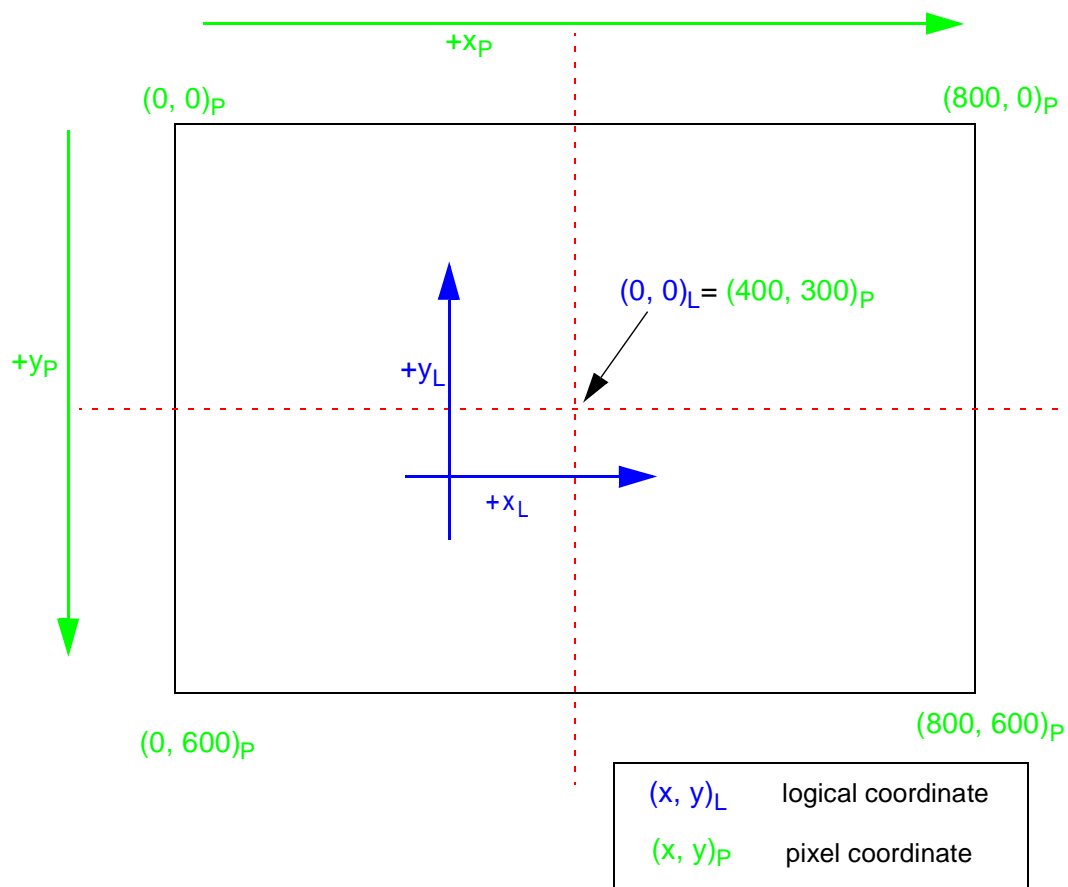
To make our discussion concrete, let's suppose the size of a `PlottingCanvas` is 800 pixels wide and 600 pixels high. Let's also suppose that the origin of the graph will be at the center of the canvas. This means that the origin point (0, 0) in the logical coordinate corresponds to the point (400, 300) in the pixel coordinate. Remember that the point (0, 0) in the pixel coordinate is the top left corner of the canvas and the `x` and `y` values increase as they move toward right and down, respectively. Figure 1 contrasts the two coordinate systems.

Now suppose we pass two logical points $L1 = (40, 80)$ and $L2 = (100, 50)$. To draw a line between these two points, we need to convert $L1$ and $L2$ to pixel coordinates. Corresponding pixel coordinates for $L1$ and $L2$ are shown in Figure 2. Given a logical point (x_L, y_L) , we can convert it to a pixel point (x_P, y_P) using the following formula:

$$x_P = OX_P + x_L;$$

FIGURE 1

The difference between the logical and pixel coordinates.



$$y_p = OY_p - y_L;$$

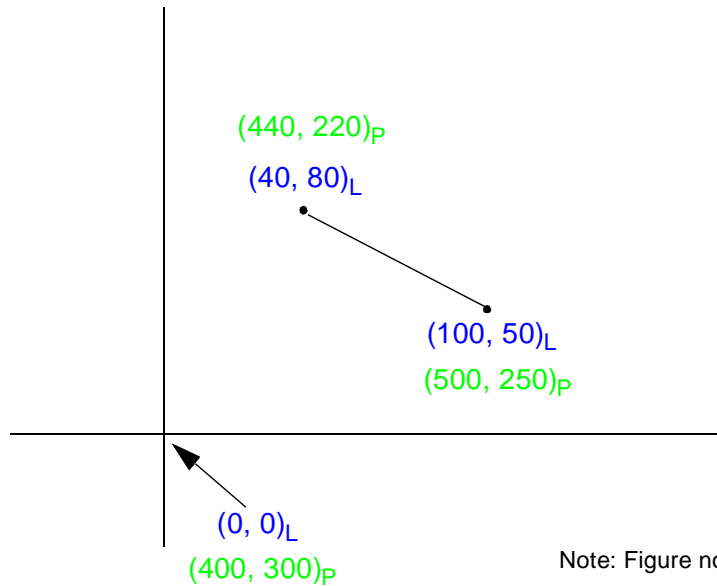
where OX_p and OY_p are x and y values of the origin point in pixel coordinates. For example, L1 is converted as follows:

$$\begin{array}{rcl} 400 & + & 40 \quad \text{--->} \quad 440 \\ 300 & - & 80 \quad \text{--->} \quad 220 \end{array}$$

The given formula assumes the default scaling of one pixel per logical unit. If it is different from the default, then we need to take it into consideration as

FIGURE 2

Mapping logical coordinates to pixel coordinates assuming one pixel per logical unit.



$$x_p = OX_p + x_L * \text{unit};$$

$$y_p = OY_p - y_L * \text{unit};$$

In the PlottingCanvas class, we use data members originX and originY to keep track of OX_p and OY_p and a data member unit to keep track of the scaling factor. Using these formulas, the plotLines method is implemented as follows:

```
private void plotLines(Graphics g)
{
    int x1, y1, x2, y2;

    Point2D.Double p1, p2;

    //draw lines in blue
    g.setColor(Color.blue);

    for (int i = 0; i < currentPt-1; i++) {
```

Type casting is necessary because **drawLine** accepts only integers.

```

        p1 = (Point2D.Double) pt.get(i);
        p2 = (Point2D.Double) pt.get(i+1);

        x1 = (int) (originX + p1.x * unit);
        y1 = (int) (originY - p1.y * unit);
        x2 = (int) (originX + p2.x * unit);
        y2 = (int) (originY - p2.y * unit);

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

```

Notice that we are retrieving every pair from the `pt` vector. This means that the same point (other than the very first and the last) is retrieved twice from the `pt` vector. You can improve the code slightly by retrieving only once. We leave it in the current form since it is easier to understand. The improvement will be left as an exercise.

The `drawAxes` method uses the `originX` and `originY` values to draw the lines along the X and Y axes in gray. Here's the method:

```

private void drawAxis(Graphics g)
{
    g.setColor(Color.gray);

    //draw the X axis
    g.drawLine(0, originY, getSize().width, originY);

    //draw the Y axis
    g.drawLine(originX, 0, originX, getSize().height);
}

```

The `getSize` method (this is an inherited method) retrieves the dimension of the canvas, and we get the width and height information from the returned dimension as `getSize().width` and `getSize().height`. The `getSize` method is used in the `setOrigin` method to convert the logical origin (0, 0) to the corresponding pixel coordinates.

The remaining methods in the class should be straightforward once you understand the logical-to-pixel coordinate mapping. We are now ready to list the complete `PlottingCanvas` class.

File: [PlottingCanvas.java](#)

```

import java.awt.*;
import java.awt.geom.*;

```

```

import java.util.*;

/**
 * Introduction to OOP with Java 2nd Edition, McGraw-Hill
 *
 * <p>
 * This class allows plotting of points in the 2D space. To use
 * a PlottingCanvas, first make a call to setStartingPoint to set
 * the starting point of a graph. Then call setNextPoint for
 * as many times as necessary to complete the graph.
 *
 * <p>
 * Internally a vector is used to keep track of all points. All
 * the recorded points are redrawn every time refreshing of
 * the canvas is necessary.
 *
 * <p>
 * For a Swing version, you need to extend JPanel instead of Canvas
 * and make necessary adjustments.
 *
 * @author Dr. Caffeine
 */
public class PlottingCanvas extends Canvas
{
    //-----
    //      Data Members
    //-----

    /**
     * Default number of pixels per logical unit
     */
    private final int    DEFAULT_UNIT = 2;

    /**
     * A counter for number points drawn
     */
    private int          currentPt;

    /**
     * A vector to keep track of all drawn points
     */
    private Vector        pt;

    /**
     * The X coordinate of the origin (0,0) of the graph
     * in pixel coordinate
     */
    private int          originX;

    /**
     * The Y coordinate of the origin (0,0) of the graph
     * in pixel coordinate
     */
    private int          originY;

    /**
     * The X coordinate of the window center in
     * logical coordinate

```

```

    */
    private int        centerX;

    /**
     * The Y coordinate of the window center in
     * logical coordinate
     */
    private int        centerY;

    /**
     * The number of pixels used for a single logical unit
     */
    private double      unit;

//-----
//      Constructors
//-----

    /**
     * Default constructor
     */
    public PlottingCanvas()
    {
        super();

        pt          = new Vector( 100, 25 );

        currentPt    = 0;

        unit         = DEFAULT_UNIT;

        setOrigin(0,0); //make the window center the graph's origin
    }

//-----
//      Public Methods:
//
//      void      paint          ( Graphics          )
//                  Do not call this method
//
//      void      setNextPoint    ( Point2D.Double      )
//      void      setOrigin       ( int,  int          )
//      void      setUnit         ( double             )
//
//-----

    /**
     * Paints the graph along with the X and Y axes.
     * @param g Graphics object to draw
     */
    public void paint(Graphics g)
    {
        drawAxis(g);
        plotLines(g);
    }

```

```

/**
 * Sets the next point in the graph. The graph is
 * redrawn whenever a new point is added by
 * calling this method.
 *
 * @param x the X coordinate of the first point in the graph
 * @param y the Y coordinate of the first point in the graph
 */
public void setNextPoint( double x, double y )
{
    pt.add( new Point2D.Double( x, y ) );

    repaint( );

    currentPt++;
}

/**
 * Sets the origin of the graph. The value (x,y) designates
 * the value in the logical coordinate that corresponds
 * to the physical center of the window.
 *
 * @param x the X coordinate of the origin point
 * @param y the Y coordinate of the origin point
 */
public void setOrigin( int x, int y )
{
    originX = (int) (getSize().width / 2 - x * unit);
    originY = (int) (getSize().height / 2 + y * unit);

    centerX = x;
    centerY = y;
}

/**
 * Sets the graph's unit - the number of pixels used
 * per single logical unit.
 *
 * @param pixelsPerUnit the number of pixels used for
 *                        a single logical unit
 */
public void setUnit(double pixelsPerUnit)
{
    unit = pixelsPerUnit;
    setOrigin(centerX, centerY);
}

//-----
//      Private Methods:
//
//      void      drawAxis      ( Graphics      )
//      void      plotLines     ( Graphics      )
//
//-----

/**

```

```

    * Draws the X and Y axes in gray.
    *
    * @param g the Graphics object where the axes are drawn
    */
private void drawAxis(Graphics g)
{
    g.setColor(Color.gray);

    //draw the X axis
    g.drawLine(0, originY, getSize().width, originY);

    //draw the Y axis
    g.drawLine(originX, 0, originX, getSize().height);
}

/**
 * Draws the graph by connecting the points in the
 * vector pt.
 *
 * @param g the Graphics object where the graph is drawn
 *
 */
private void plotLines(Graphics g)
{
    int x1, y1, x2, y2;

    Point2D.Double p1, p2;

    //draw lines in blue
    g.setColor(Color.blue);

    for (int i = 0; i < currentPt-1; i++) {

        p1 = (Point2D.Double) pt.get(i);
        p2 = (Point2D.Double) pt.get(i+1);

        x1 = (int) (originX + p1.x * unit);
        y1 = (int) (originY - p1.y * unit);
        x2 = (int) (originX + p2.x * unit);
        y2 = (int) (originY - p2.y * unit);

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

    }
}
}

```

3 On Your Own

There are number of things you can do to make the PlottingCanvas more useful. One is the drawing of grid lines, say, every 50 logical units or a programmer specified interval. Another is the drawing of tick marks on the axes. The third is the drawing of the graph in a different line thickness and color.

In the sample programs we presented here, the values are computed within the programs. It is very useful to have a program that reads the data from a file and plot the graph. For instance, you may use a program written in other pro-

gramming languages or use a commercial software that saves the points in the graph in a file. You can write a short program to read these data in a file and plot the graph using a `PlottingCanvas`. Chapter 11 and Special Topics No. 7 discuss the file I/O.