```java
import java.awt.Color;

/**
 * A library of image processing functions.
 */
public class Instush {

    public static void main(String[] args) {
        Color[][] image = read("xmen.ppm");
        //print(image);
        // Color[][] image = read("ironman.ppm");
        // print(image);
        // show(image);
        // Color[][] flippedHorz = flippedHorizontally(image);
        // print(flippedHorz);
        // show(flippedHorz);
        // Color[][] flippedVert = flippedVertically(image);
        // show(flippedVert);
        // print(flippedVert);
        // Color[][] grayedImage = greyscaled(image);
        // print(grayedImage);
        // Color[][] scaleddImage = scaled(image, 3, 5);
        // print(scaleddImage);
        // Color[][] belndedImage = blend(image, image2, 0.25);
        // print(belndedImage);
        // Can be used for testing, as needed.
    }

    /**
     * Returns an image created from a given PPM file.
     * SIDE EFFECT: Sets standard input to the given file.
     * @return the image, as a 2D array of Color values
     */
    public static Color[][] read(String filename) {
        StdIn.setInput(filename);
        // Reads the PPM file header (ignoring some items)
        StdIn.readString();
        int numCols = StdIn.readInt();
        int numRows = StdIn.readInt();
        StdIn.readInt();
        // Creates the image
        Color[][] image = new Color[numRows][numCols];
        for (int rows = 0; rows < numRows; rows++) {
            int r = -1;
            int g = -1;
            int b = -1;
            for (int cols = 1; cols <= numCols * 3; cols++) {
                int rgbNum = StdIn.readInt();
                if (r == -1) {
                    r = rgbNum;
                } else if (g == -1) {
                    g = rgbNum;
                } else {
                    b = rgbNum;
                    image[rows][cols / 3 - 1] = new Color(r, g, b);
                    r = -1;
                    g = -1;
                    b = -1;
                }
            }
        }
        // Reads the RGB values from the file, into the image.
        // For each pixel (i,j), reads 3 values from the file,
        // creates from the 3 colors a new Color object, and
        // makes pixel (i,j) refer to that object.
        return image;
    }

    /**
     * Prints the pixels of a given image.
     * Each pixel is printed as a triplet of (r,g,b) values.
     * For debugging purposes.
     * @param image - the image to be printed
     */
    public static void print(Color[][] image) {
        int numRows = image.length;
        int numCols = image[0].length;
        for (int rows = 0; rows < numRows; rows++) {
            for (int cols = 0; cols < numCols; cols++) {
                System.out.print("(");
                System.out.printf("%3s" + ",", image[rows][cols].getRed());    // Prints the color's red component
                System.out.printf("%4s" + ",", image[rows][cols].getGreen());  // Prints the color's green component
                System.out.printf("%4s", image[rows][cols].getBlue());   // Prints the color's blue component
                System.out.print(") ");
            }
        }
        System.out.println("");
        System.out.println("");
    }

    /**
     * Returns an image which is the horizontally flipped version of the given image.
     * @param image - the image to flip
     * @return the horizontally flipped image
     */
    public static Color[][] flippedHorizontally(Color[][] image) {
        int numRows = image.length;
        int numCols = image[0].length;
        Color[][] flippedImage = new Color[numRows][numCols];
        for (int rows = 0; rows < numRows; rows++) {
            for (int cols = 0; cols < numCols; cols++) {
                flippedImage[rows][cols] = image[rows][(numCols - 1) - cols];
            }
        }

        return flippedImage;
    }

    /**
     * Returns an image which is the vertically flipped version of the given image.
     * @param image - the image to flip
     * @return the vertically flipped image
     */
    public static Color[][] flippedVertically(Color[][] image){
        int numRows = image.length;
        int numCols = image[0].length;

        Color[][] flippedImage = new Color[numRows][numCols];
        for (int rows = 0; rows < numRows; rows++) {
            for (int cols = 0; cols < numCols; cols++) {
                flippedImage[rows][cols] = image[(numRows -1) - rows][cols];
            }
        }
        return flippedImage;
    }

    /**
     * Returns the average of the RGB values of all the pixels in a given image.
     * @param image - the image
     * @return the average of all the RGB values of the image
     */
    public static double average(Color[][] image) {
        return 0.0;
    }

    /**
     * Returns the luminance value of a given pixel. Luminance is a weighted average
     * of the RGB values of the pixel, given by 0.299 * r + 0.587 * g + 0.114 * b.
     * Used as a shade of grey, as part of the greyscaling process.
     * @param pixel - the pixel
     * @return the greyscale value of the pixel, as a Color object
     *         (r = g = b = the greyscale value)
     */
    public static Color luminance(Color pixel) {
        double grayValue = Math.floor((pixel.getRed() * 0.299) + (pixel.getGreen() * 0.587) + (pixel.getBlue() * 0.114));
        int IntValue = (int) grayValue;
        Color grayedPixel = new Color(Math.round(IntValue), Math.round(IntValue), Math.round(IntValue));

        return grayedPixel;
    }

    /**
     * Returns an image which is the greyscaled version of the given image.
     * @param image - the image
     * @return rhe greyscaled version of the image
     */
    public static Color[][] greyscaled(Color[][] image) {
        int numRows = image.length;
        int numCols = image[0].length;
        Color[][] grayScaledImage = new Color[numRows][numCols];
        for (int rows = 0; rows < numRows; rows++) {
            for (int cols = 0; cols < numCols; cols++) {
                grayScaledImage[rows][cols] = luminance(image[rows][cols]);
            }
        }
        return grayScaledImage;
    }

    /**
     * Returns an umage which is the scaled version of the given image.
     * The image is scaled (resized) to be of the given width and height.
     * @param image - the image
     * @param width - the width of the scaled image
     * @param height - the height of the scaled image
     * @return - the scaled image
     */
    public static Color[][] scaled(Color[][] image, int width, int height) {
        int w0 = image[0].length;
        int h0 = image.length;
        int W = width;
        int H = height;
        Color[][] scaledImage = new Color[H][W];

        for (int rows = 0; rows < H; rows++) {
            for (int cols = 0; cols < W; cols++) {
                scaledImage[rows][cols] = image[(int)Math.floor(rows * (double)h0 / (double)H)][(int)Math.floor(cols * (double)w0 / (double)W)];
            }
        }

        return scaledImage;
    }

    /**
     * Returns a blended color which is the linear combination of two colors.
     * Each r, g, b, value v is calculated using v = (1 - alpha) * v1 + alpha * v2.
     *
     * @param pixel1 - the first color
     * @param pixel2 - the second color
     * @param alpha - the linear combination parameter
     * @return the blended color
     */
    public static Color blend(Color c1, Color c2, double alpha) {
        int newRed = (int)(alpha * c1.getRed() + (1 - alpha) * c2.getRed());
        int newGreen = (int)(alpha * c1.getGreen() + (1 - alpha) * c2.getGreen());
        int newBlue = (int)(alpha * c1.getBlue() + (1 - alpha) * c2.getBlue());
        Color blendedColor = new Color(newRed, newGreen, newBlue);

        return blendedColor;
    }

    /**
     * Returns an image which is the blending of the two given images.
     * The blending is the linear combination of (1 - alpha) parts the
     * first image and (alpha) parts the second image.
     * The two images must have the same dimensions.
     * @param image1 - the first image
     * @param image2 - the second image
     * @param alpha - the linear combination parameter
     * @return - the blended image
     */
    public static Color[][] blend(Color[][] image1, Color[][] image2, double alpha) {
        int numRows = image1.length;
        int numCols = image1[0].length;
        Color[][] blendedImage = new Color[numRows][numCols];
        for (int rows = 0; rows < numRows; rows++) {
            for (int cols = 0; cols < numCols; cols++) {
                blendedImage[rows][cols] = blend(image1[rows][cols], image2[rows][cols], alpha);
            }
        }
        return blendedImage;
    }

    /**
     * Morphs the source image into the target image, gradually, in n steps.
     * Animates the morphing process by displaying the morphed image in each step.
     * The target image is an image which is scaled to be a version of the target
     * image, scaled to have the width and height of the source image.
     * @param source - source image
     * @param target - target image
     * @param n - number of morphing steps
     */
    public static void morph(Color[][] source, Color[][] target, int n) {
        int i = 0;
        int sourceW = source[0].length;
        int sourceH = source.length;;
        Color[][] scaledTargetImage = scaled(target, sourceW, sourceH);
        while (i < n) {
            double alpha = (double)(n - i) / n;
            show(blend(source, scaledTargetImage, alpha));
            i++;
        }
    }

     /**
     * Renders (displays) an image on the screen, using StdDraw.
     *
     * @param image - the image to show
     */
    public static void show(Color[][] image) {
        StdDraw.setCanvasSize(image[0].length, image.length);
        int width = image[0].length;
        int height = image.length;
        StdDraw.setXscale(0, width);
        StdDraw.setYscale(0, height);
        StdDraw.show(25);
        for (int i = 0; i < height; i++) {
            for (int j = 0; j < width; j++) {
                // Sets the pen color to the color of the pixel
                StdDraw.setPenColor( image[i][j].getRed(),
                                     image[i][j].getGreen(),
                                     image[i][j].getBlue() );
                // Draws the pixel as a tiny filled square of size 1
                StdDraw.filledSquare(j + 0.5, height - i - 0.5, 0.5);
            }
        }
        StdDraw.show();
    }
}
```