

```
60 import java.awt.Color;
61
62 /**
63  * Demonstrates three Instush.java services: flipping
64  * an image horizontally, flipping an image
65  * vertically, and greyscaling an image.
66  *
67  * The program recieves two command-line arguments:
68  * the name of the PPM file that represents
69  * the source image (a string), and one of the strings
70  * "fh", "fv", or "gs" (a string). The program
71  * creates and displays a new image which is either
72  * the horizontally flipped version of the source
73  * image ("fh"), or the vertically flipped version of
74  * the source image ("fv"), or the greyscaled
75  * version of the source image ("gs"). For example:
76  * java Editor1 thor.ppm gs
77  */
78 public class Editor1 {
79
80     public static void main (String[] args){
81         String fileName = args[0];
82         String action = args[1];
83         Color[][] image = Instush.read(fileName);
84         if (action.equals("fh")) {
85             Instush.show(Instush.flippedHorizontally(image));
86         } else if (action.equals("fv")) {
87             Instush.show(Instush.flippedVertically(image));
88         } else if (action.equals("gs")) {
89             //gs
90             Instush.show(Instush.greyscaled(image));
91         }
92     }
93 }
```

```
60 import java.awt.Color;
61
62 /**
63  * Demonstrates the scaling function of Instush.java.
64  * The program receives two command-line arguments:
65  * the name of the PPM file
66  * (a string) representing the image that should be
67  * scaled, and two integers
68  * that specify the width and the height of the scaled
69  * image. For example:
70  * java Editor2 ironman.ppm 100 900
71  */
72 public class Editor2 {
73
74     public static void main (String[] args){
75         String fileName = args[0];
76         int newScaleX = Integer.parseInt(args[1]);
77         int newScaleY = Integer.parseInt(args[2]);
78         Color[][] image = Instush.read(fileName);
79         Instush.show(Instush.scaled(image, newScaleX,
80 newScaleY));
81     }
82 }
```

```
1 import java.awt.Color;
2
3 /**
4  * Demonstrates the morphing service of Instush.java.
5  * The program recieves three command-line arguments:
6  * the name of a PPM file
7  * that represents the source image (a string), the
8  * name of a PPM file that represents
9  * the target image (a string), and the number of
10 * morphing steps (an int).
11 * For example:
12 * java Editor3 cake.ppm ironman.ppm 300
13 * If the two images don't have the same dimensions,
14 * the program scales the target image
15 * to the dimensions of the source image.
16 */
17 public class Editor3 {
18
19     public static void main (String[] args) {
20         String fileName1 = args[0];
21         String fileName2 = args[1];
22         int numOfSteps = Integer.parseInt(args[2]);
23         Color[][] image = Instush.read(fileName1);
24         Color[][] image2 = Instush.read(fileName2);
25         Instush.morph(image, image2, numOfSteps);
26     }
27 }
```

```
1 import java.awt.Color;
2
3 /**
4  * Demonstrates the morphing service of Instush.java
5  * with a grayscale image of the source.
6  * The program receives two command-line arguments:
7  * the name of a PPM file
8  * that represents the source image (a string) and the
9  * number of morphing steps (an int).
10  * For example:
11  * java Editor4 ironman.ppm 300
12  */
13 public class Editor4 {
14
15     public static void main (String[] args) {
16         String fileName1 = args[0];
17         int numOfSteps = Integer.parseInt(args[1]);
18         Color[][] image = Instush.read(fileName1);
19
20         Instush.morph(image,
21 Instush.greyscaled(image), numOfSteps);
22     }
23 }
```

```
1 import java.awt.Color;
2
3 /**
4  * A library of image processing functions.
5  */
6 public class Instush {
7
8     public static void main(String[] args) {
9         Color[][] image = read("xmen.ppm");
10        //print(image);
11        // Color[][] image = read("ironman.ppm");
12        // print(image);
13        // show(image);
14        // Color[][] flippedHorz = flippedHorizontally(image);
15        // print(flippedHorz);
16        // show(flippedHorz);
17        // Color[][] flippedVert = flippedVertically(image);
18        // show(flippedVert);
19        // print(flippedVert);
20        // Color[][] grayedImage = greyscaled(image);
21        // print(grayedImage);
22        // Color[][] scaledImage = scaled(image, 3, 5);
23        // print(scaledImage);
24        // Color[][] blendedImage = blend(image, image2, 0.25);
25        // print(blendedImage);
26        // Can be used for testing, as needed.
27    }
28
29    /**
30     * Returns an image created from a given PPM file.
31     * SIDE EFFECT: Sets standard input to the given file.
32     * @return the image, as a 2D array of Color values
33     */
34    public static Color[][] read(String filename) {
35        StdIn.setInput(filename);
36        // Reads the PPM file header (ignoring some items)
37        StdIn.readString();
38        int numCols = StdIn.readInt();
39        int numRows = StdIn.readInt();
40        StdIn.readInt();
41        // Creates the image
42        Color[][] image = new Color[numRows][numCols];
43        for (int rows = 0; rows < numRows; rows++) {
44            int r = -1;
45            int g = -1;
46            int b = -1;
47            for (int cols = 0; cols < numCols; cols++) {
48                int rgbNum = StdIn.readInt();
49                if (r == -1) {
50                    r = rgbNum;
51                } else if (g == -1) {
52                    g = rgbNum;
53                } else {
54                    b = rgbNum;
55                    image[rows][cols] = new Color(r, g, b);
56                    r = -1;
57                    g = -1;
58                    b = -1;
59                }
60            }
61        }
62    }
63 }
```

```

62     // Reads the RGB values from the file, into the image.
63     // For each pixel (i,j), reads 3 values from the file,
64     // creates from the 3 colors a new Color object, and
65     // makes pixel (i,j) refer to that object.
66     return image;
67 }
68
69 /**
70  * Prints the pixels of a given image.
71  * Each pixel is printed as a triplet of (r,g,b) values.
72  * For debugging purposes.
73  * @param image - the image to be printed
74  */
75 public static void print(Color[][] image) {
76     int numRows = image.length;
77     int numCols = image[0].length;
78     for (int rows = 0; rows < numRows; rows++) {
79         for (int cols = 0; cols < numCols; cols++) {
80             System.out.print("(");
81             System.out.printf("%3s" + ",", image[rows][cols].getRed());    // Prints the color's red component
82             System.out.printf("%4s" + ",", image[rows][cols].getGreen()); // Prints the color's green component
83             System.out.printf("%4s", image[rows][cols].getBlue());    // Prints the color's blue component
84             System.out.print(") ");
85         }
86     }
87     System.out.println("");
88     System.out.println("");
89 }
90
91 /**
92  * Returns an image which is the horizontally flipped version of the given
93  * image.
94  * @param image - the image to flip
95  * @return the horizontally flipped image
96  */
97 public static Color[][] flippedHorizontally(Color[][] image) {
98     int numRows = image.length;
99     int numCols = image[0].length;
100    Color[][] flippedImage = new Color[numRows][numCols];
101    for (int rows = 0; rows < numRows; rows++) {
102        for (int cols = 0; cols < numCols; cols++) {
103            flippedImage[rows][cols] = image[rows][(numCols - 1) - cols];
104        }
105    }
106    return flippedImage;
107 }
108
109 /**
110  * Returns an image which is the vertically flipped version of the given image.
111  * @param image - the image to flip
112  * @return the vertically flipped image
113  */
114 public static Color[][] flippedVertically(Color[][] image){
115     int numRows = image.length;
116     int numCols = image[0].length;
117
118     Color[][] flippedImage = new Color[numRows][numCols];
119     for (int rows = 0; rows < numRows; rows++) {

```

```

120         for (int cols = 0; cols < numCols; cols++) {
121             flippedImage[rows][cols] = image[(numRows - 1) - rows][cols];
122         }
123     }
124     return flippedImage;
125 }
126
127 /**
128  * Returns the average of the RGB values of all the pixels in a given image.
129  * @param image - the image
130  * @return the average of all the RGB values of the image
131  */
132 public static double average(Color[][] image) {
133     return 0.0;
134 }
135
136 /**
137  * Returns the luminance value of a given pixel. Luminance is a weighted
average
138  * of the RGB values of the pixel, given by  $0.299 * r + 0.587 * g + 0.114 * b$ .
139  * Used as a shade of grey, as part of the greyscaling process.
140  * @param pixel - the pixel
141  * @return the greyscale value of the pixel, as a Color object
142  *         (r = g = b = the greyscale value)
143  */
144 public static Color luminance(Color pixel) {
145     double grayValue = Math.floor((pixel.getRed() * 0.299) + (pixel.getGreen()
* 0.587) + (pixel.getBlue() * 0.114));
146     int IntValue = (int) grayValue;
147     Color grayedPixel = new Color(Math.round(IntValue), Math.round(IntValue),
Math.round(IntValue));
148
149     return grayedPixel;
150 }
151
152 /**
153  * Returns an image which is the greyscaled version of the given image.
154  * @param image - the image
155  * @return the greyscaled version of the image
156  */
157 public static Color[][] greyscaled(Color[][] image) {
158     int numRows = image.length;
159     int numCols = image[0].length;
160     Color[][] grayScaledImage = new Color[numRows][numCols];
161     for (int rows = 0; rows < numRows; rows++) {
162         for (int cols = 0; cols < numCols; cols++) {
163             grayScaledImage[rows][cols] = luminance(image[rows][cols]);
164         }
165     }
166     return grayScaledImage;
167 }
168
169 /**
170  * Returns an image which is the scaled version of the given image.
171  * The image is scaled (resized) to be of the given width and height.
172  * @param image - the image
173  * @param width - the width of the scaled image
174  * @param height - the height of the scaled image
175  * @return the scaled image
176  */
177 public static Color[][] scaled(Color[][] image, int width, int height) {
178     int w0 = image[0].length;

```

```

179     int h0 = image.length;
180     int W = width;
181     int H = height;
182     Color[][] scaledImage = new Color[H][W];
183
184     for (int rows = 0; rows < H; rows++) {
185         for (int cols = 0; cols < W; cols++) {
186             scaledImage[rows][cols] = image[(int)Math.floor(rows * (double)h0 /
(double)H)][(int)Math.floor(cols * (double)w0 / (double)W)];
187         }
188     }
189
190     return scaledImage;
191 }
192
193 /**
194  * Returns a blended color which is the linear combination of two colors.
195  * Each r, g, b, value v is calculated using  $v = (1 - \alpha) * v1 + \alpha * v2$ .
196  *
197  * @param pixel1 - the first color
198  * @param pixel2 - the second color
199  * @param alpha - the linear combination parameter
200  * @return the blended color
201  */
202 public static Color blend(Color c1, Color c2, double alpha) {
203     int newRed = (int)(alpha * c1.getRed() + (1 - alpha) * c2.getRed());
204     int newGreen = (int)(alpha * c1.getGreen() + (1 - alpha) * c2.getGreen());
205     int newBlue = (int)(alpha * c1.getBlue() + (1 - alpha) * c2.getBlue());
206     Color blendedColor = new Color(newRed, newGreen, newBlue);
207
208     return blendedColor;
209 }
210
211 /**
212  * Returns an image which is the blending of the two given images.
213  * The blending is the linear combination of (1 - alpha) parts the
214  * first image and (alpha) parts the second image.
215  * The two images must have the same dimensions.
216  * @param image1 - the first image
217  * @param image2 - the second image
218  * @param alpha - the linear combination parameter
219  * @return - the blended image
220  */
221 public static Color[][] blend(Color[][] image1, Color[][] image2, double alpha)
{
222     int numRows = image1.length;
223     int numCols = image1[0].length;
224     Color[][] blendedImage = new Color[numRows][numCols];
225     for (int rows = 0; rows < numRows; rows++) {
226         for (int cols = 0; cols < numCols; cols++) {
227             blendedImage[rows][cols] = blend(image1[rows][cols], image2[rows]
[cols], alpha);
228         }
229     }
230     return blendedImage;
231 }
232
233 /**
234  * Morphs the source image into the target image, gradually, in n steps.
235  * Animates the morphing process by displaying the morphed image in each step.
236  * The target image is an image which is scaled to be a version of the target
237  * image, scaled to have the width and height of the source image.

```



```
238 * @param source - source image
239 * @param target - target image
240 * @param n - number of morphing steps
241 */
242 public static void morph(Color[][] source, Color[][] target, int n) {
243     int i = 0;
244     int sourceW = source[0].length;
245     int sourceH = source.length;
246     Color[][] scaledTargetImage = scaled(target, sourceW, sourceH);
247     while (i < n) {
248         double alpha = (double)(n - i) / n;
249         show(blend(source, scaledTargetImage, alpha));
250         i++;
251     }
252 }
253
254 /**
255  * Renders (displays) an image on the screen, using StdDraw.
256  *
257  * @param image - the image to show
258  */
259 public static void show(Color[][] image) {
260     StdDraw.setCanvasSize(image[0].length, image.length);
261     int width = image[0].length;
262     int height = image.length;
263     StdDraw.setXscale(0, width);
264     StdDraw.setYscale(0, height);
265     StdDraw.show(25);
266     for (int i = 0; i < height; i++) {
267         for (int j = 0; j < width; j++) {
268             // Sets the pen color to the color of the pixel
269             StdDraw.setPenColor( image[i][j].getRed(),
270                                 image[i][j].getGreen(),
271                                 image[i][j].getBlue() );
272             // Draws the pixel as a tiny filled square of size 1
273             StdDraw.filledSquare(j + 0.5, height - i - 0.5, 0.5);
274         }
275     }
276     StdDraw.show();
277 }
278 }
```