1. Include Webcam and Pile libraries, as well as standard Java libraries. Webcam library can be found at `https://github.com/sarxos/webcam-capture/releases` and Pile library can be found here `https://github.com/Battleroid/Pile/releases`.

```
import java.awt.*;
import java.awt.image.BufferedImage;
import java.awt.image.ColorConvertOp;
import java.io.File;
import java.io.IOException;
import com.caseyweed.Pile;
import com.github.sarxos.webcam.Webcam;
import com.github.sarxos.webcam.WebcamResolution;
import javax.imageio.ImageIO;
```

2. Initialize the camera and set the resolution to VGA (640x480).

```
Webcam cam = Webcam.getDefault();
cam.setViewSize(WebcamResolution.VGA.getSize());
```

3. Take a photo, first by opening the image, then capturing it into a new BufferedImage, then closing the camera.

```
cam.open();
BufferedImage sample = cam.getImage();
cam.close();
```

4. In order to convert the new image to grayscale, we must use the ColorConvertOp to transform the image to the same colorspace as a new BufferedImage with the type of Gray. When finished we will have a new gray image from our color image.

```
BufferedImage gray = new BufferedImage(sample.getWidth(null),
    sample.getHeight(null), BufferedImage.TYPE_BYTE_GRAY);
ColorConvertOp op = new ColorConvertOp(sample.getColorModel().getColorSpace(),
    gray.getColorModel().getColorSpace(), null);
op.filter(sample, gray);
```

5. To write our modifications of the gray image to a new image we create an empty BufferedImage of the same type and dimensions of our gray image.

```
BufferedImage manipulated = new BufferedImage(gray.getWidth(null),
    gray.getHeight(null), BufferedImage.TYPE_BYTE_GRAY);
```

6. Using two for loops we can cycle through each row and col of the image, effectively cycling through each pixel of the source image.

```
for (int row = 0; row < gray.getWidth(); row++) {
    for (int col = 0; col < gray.getHeight(); col++) {
```

7. We will then use `gray.getRGB(row, col)` to get the four byte integer value of that particular pixel. We will then feed that value into our method `toARGB()` to get the values for each channel of the pixel (in this case Alpha, Red, Blue, and Green).

```
public static int[] toARGB(int argb) {
    int a = (argb >> 24) & 0xFF;
    int r = (argb >> 16) & 0xFF;
    int g = (argb >> 8) & 0xFF;
    int b = (argb & 0xFF);
    return new int[] {a, r, g, b};
}

int[] argb = toARGB(gray.getRGB(row, col));
```

8. Using the original values we can create a new integer array and modify the values of pixel's channels.

```
int[] inverted = new int[] {
    argb[0],
    255 - argb[1],
    255 - argb[2],
    255 - argb[3]
};
```

9. We then use another method to reverse the process to return our four channels to a single four byte integer.

```
public static int toColor(int[] rgb) {
    return (rgb[0] << 24) | (rgb[1] << 16) | (rgb[2] << 8) | rgb[3];
}

int invertedCol = toColor(inverted);
```

10. We can then set the value of our manipulated image's pixel using this new integer.

```
manipulated.setRGB(row, col, invertedCol);
```

11. Now that we have our two images, we can then use them in a Pile object to stitch the results together. Pile takes an array of BufferedImages, so create a new array of BufferedImages using our gray image and manipulated image.

```
BufferedImage[] images = new BufferedImage[] {
    gray,
    manipulated
};
```

12. Now we can create a new Pile object with an explicit grid size (2x2), our image list, and `true` to maintain the aspect ratio. The length of each grid cell size will be determined by the longest side of images in the list provided.

    ```
    Pile p = new Pile(2, 2, images, true);
    ```

13. Lastly, we can save the resulting image from our Pile object to a new file using `savePile()`.

    ```
    p.savePile("pile_comparison.png");
    ```