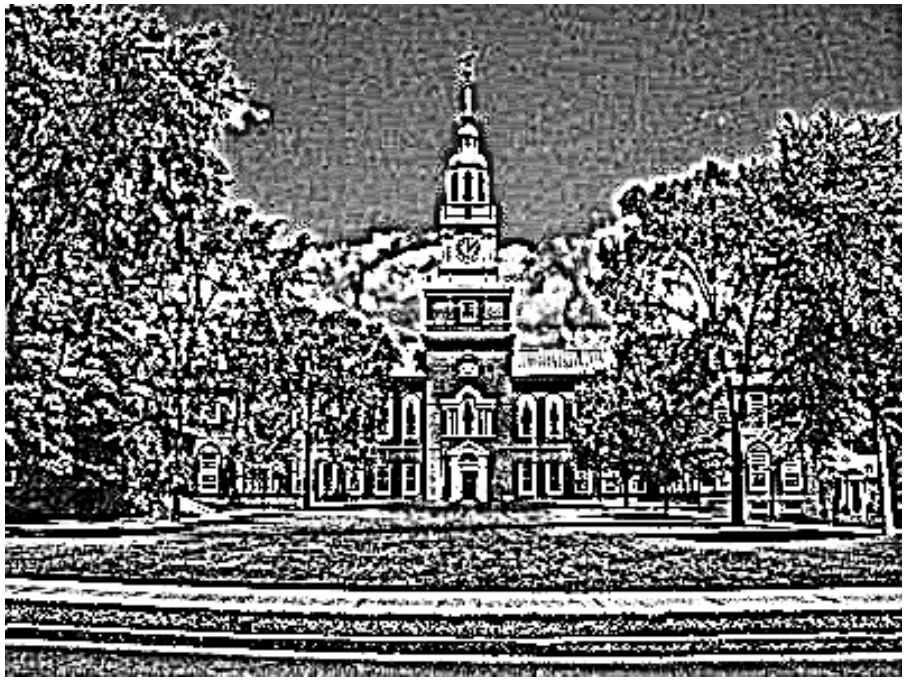
CS 10:
Problem solving via Object Oriented Programming

Image Processing

Today we will look at processing images as a step toward more sophisticated OOP

Turn this image…

… into this image



Today we will create a bare bones Photoshop

Agenda

1. Image Processing Architecture
2. Manipulating individual pixels
3. Accounting for geometry
4. Puzzle

We will create an ImageProcessor class that has methods to manipulate images

Architecture

ImageProcessor Class

Instance variables

BufferedImage image

Methods

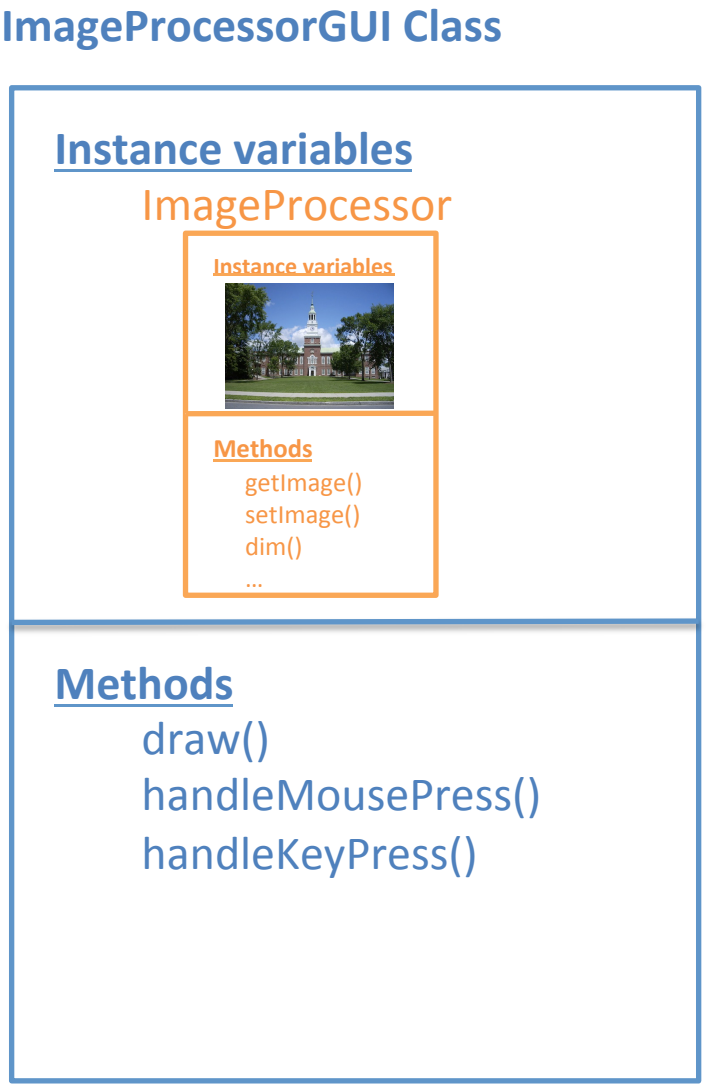
getImage()
setImage()
dim()
brighten()
average()
…

ImageProcessor Class

- Specialized for image manipulation
- Stores BufferedImage as instance variable called *image*
- Methods act on BufferedImage *image* to alter its visual appearance
- Does *not* provide user interface logic such as dealing with mouse and key presses
- Does *not* even display the image!
- We will use a separate driver app to display the image and respond to events
- *Does* provide methods to manipulate the image (ex., make brighter)

Separate ImageProcessorGUI Class will handle displaying the image and user input

Architecture



ImageProcessorGUI Class

- Driver application, specialized to create graphical user interface and respond to user input
- Has ImageProcessor as instance variable
- Methods are aimed at understanding user's intent, then calling appropriate methods on ImageProcessor to carry out intentions

Key point: Instance variables can be complex objects themselves, they aren't limited to simple things like doubles, ints, etc.

ImageProcessor0.java: Starts simple, just holds image and has getter/setter

ImageProcessor0.java

```
11 public class ImageProcessor0 {
12     private BufferedImage image;     // the current image being processed
13
14     /**
15      * @param image      the original
16      */
17     public ImageProcessor0(BufferedImage image) {
18         this.image = image;
19     }
20
21     public BufferedImage getImage() {
22         return image;
23     }
24
25     public void setImage(BufferedImage image) {
26         this.image = image;
27     }
28 }
```

- **Nothing new or fancy here**
- **Getter/setter for instance variable *image***
- **Will soon add methods to manipulate image (ex. make it brighter)**

Key point: ImageProcessor stores a BufferImage called *image* and provides methods to get/set/alter *image*

ImageProcessorGUI0.java is very similar to SimleGUI.java from last class

ImageProcessingGUI0.java

```
14 public class ImageProcessingGUI0 extends DrawingGUI {
15     private ImageProcessor0 proc;       // handles the image processing
16
17     /**
18      * Creates the GUI for the image processor, with the window scaled to the to-process image's size
19      */
20     public ImageProcessingGUI0(ImageProcessor0 proc) {
21         super("Image processing", proc.getImage().getWidth(), proc.getImage().getHeight());
22         this.proc = proc;
23     }
24
25     /**
26      * DrawingGUI method, here showing the current image
27      */
28     @Override
29     public void draw(Graphics g) {
30         g.drawImage(proc.getImage(), 0, 0, null);
31
32     }
33
34     /**
35      * DrawingGUI method, here dispatching on image processing operations
36      */
37     @Override
38     public void handleKeyPress(char op) {
39         System.out.println("Handling key '"+op+"'");
```

- **Instead of holding image like SimleGUI.java does, this class holds an *ImageProcessor***
- ***ImageProcessor* responsible for manipulating image**
- **Same *draw()* method as SmileGUI.java but now image comes from *ImageProcessor***
- **Image retrieved from *ImageProcessor* via *getImage()* (shown on previous page)**

ImageProcessorGUI0.java is very similar to SimleGUI.java from last class

ImageProcessingGUI0.java

```
28      @Override
29      public void draw(Graphics g) {
30          g.drawImage(proc.getImage(), 0, 0, null);
31
32      }
33
34      /**
35       * DrawingGUI method, here dispatching on image processing operations
36       */
37      @Override
38      public void handleKeyPress(char op) {
39          System.out.println("Handling key '"+op+"'");
40          if (op=='s') { // save a snapshot
41              saveImage(proc.getImage(), "pictures/snapshot.png", "png");
42          }
43          else {
44              System.out.println("Unknown operation");
45          }
46
47          repaint(); // Re-draw, since image has changed
48      }
49
50      public static void main(String[] args) {
51          SwingUtilities.invokeLater(new Runnable() {
52              public void run() {
53                  // Load the image to process
54                  BufferedImage baker = loadImage("pictures/baker.jpg");
55                  // Create a new processor, and a GUI to handle it
56                  new ImageProcessingGUI0(new ImageProcessor0(baker));
57              }
58          });
59      }
60  }
```

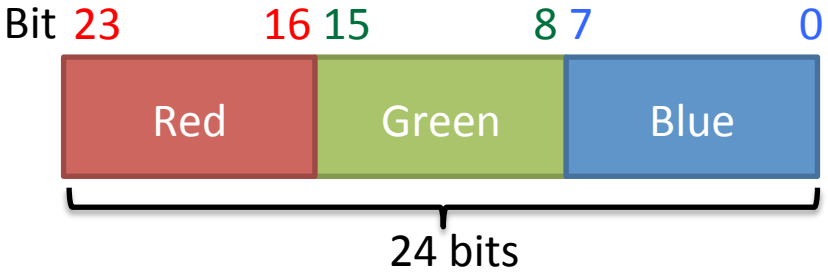
- **Can save image by pressing 's' on keyboard**
- **Calls DrawingGUI method *saveImage()***
- **NOTE: Eclipse won't show new file until File->Refresh**

- ***main()* creates a new ImageProcessor0 and provides the constructor with an image**
- ***main()* passes the new ImageProcessor0 to its own constructor**

Agenda

1. Image Processing Architecture
2. Manipulating individual pixels
3. Accounting for geometry
4. Puzzle

Pixel colors are made up of Red, Green, and Blue components of varying intensity



Each R,G, or B components has 8 bits to control color intensity

8 bits means intensity range 0-255

Red	Green	Blue	Result	
255	255	255	**White**	All colors full on
0	0	0	**Black**	All colors off
255	0	0	**Bright red**	One color full on, others off
0	255	0	**Bright green**	
0	0	255	**Bright blue**	
128	0	0	**Not-as-bright-red**	One color half on, others off
0	128	0	**Not-as-bright green**	
0	0	128	**Not-as-bright-blue**	

We can pick up the color of a pixel, modify it, and write it back to the image

ColorDim.java

```
 9 public class ColorDim {
10
11     public static void main(String[] args) {
12         //load an image
13         BufferedImage image = DrawingGUI.loadImage("pictures/baker.jpg");
14         int x=0, y=0;
15
16         //pick up color of pixel at x,y location
17         Color color = new Color(image.getRGB(x,y));
18         System.out.println("Original color "+ color);
19
20         //extract red, green and blue components
21         //divide by 2 to dim them
22         int red = color.getRed()/2;
23         int green = color.getGreen()/2;
24         int blue = color.getBlue()/2;
25
26         //write dimmed color back to image
27         Color newColor = new Color(red, green, blue);
28         System.out.println("New color " + newColor);
29         image.setRGB(x, y, newColor.getRGB());
30     }
31 }
32
```

Load image

Get color at x,y

Dim color by dividing R, G, and B components by 2

Create new dimmed color and write it back to the image

```
Problems  Javadoc  Declaration  Console  Debug  Expressions  Error Log  Call Hierarchy
<terminated> ColorDim [Java Application] /Library/Java/JavaVirtualMachines/jdk1.8.0_112.jdk/Contents/Home/bin/java (Jan 9, 2018, 10:38:18 AM)
Original color java.awt.Color[r=81,g=109,b=51]
New color java.awt.Color[r=40,g=54,b=25]
```

Similarly, we can dim all pixels in an image with a nested loop

ImageProcessor.java

```
/**
 * Dims the current image by scaling down pixel values.
 */
public void dim() {
    // Nested loop over every pixel
    for (int y = 0; y < image.getHeight(); y++) {
        for (int x = 0; x < image.getWidth(); x++) {
            // Get current color; scale each channel; put new color
            Color color = new Color(image.getRGB(x, y));
            int red = color.getRed() * 3/4;
            int green = color.getGreen() * 3/4;
            int blue = color.getBlue() * 3/4;
            Color newColor = new Color(red, green, blue);
            image.setRGB(x, y, newColor.getRGB());
        }
    }
}
```

C-style for loop:
for (init;condition;increment)
Initialize y to 0
Loop while y < height
Increment y after each loop

Similarly, we can dim all pixels in an image with a nested loop

ImageProcessor.java

```
/**
 * Dims the current image by scaling down pixel values.
 */
public void dim() {
    // Nested loop over every pixel
    for (int y = 0; y < image.getHeight(); y++) {
        for (int x = 0; x < image.getWidth(); x++) {
            // Get current color; scale each channel; put new color
            Color color = new Color(image.getRGB(x, y));
            int red = color.getRed() * 3/4;
            int green = color.getGreen() * 3/4;
            int blue = color.getBlue() * 3/4;
            Color newColor = new Color(red, green, blue);
            image.setRGB(x, y, newColor.getRGB());
        }
    }
}
```

Loop over every *y* value

Loop over every *x* value

Together a "nested loop"

Will cover all *x,y* locations

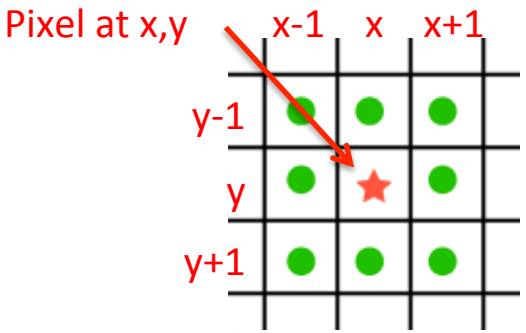
Here we scale color intensity by a factor of 75%

Write changes back to *image*

Remember: *image* is an instance variable holding the image we are manipulating

We can also focus on a portion of the screen with a nested loop

ImageProcessor.java



Draw a square around the mouse location of given *color*

```
291    /**
292     * Updates the image with a square at the location, colored the given color.
293     *
294     * @param cx   center x of square
295     * @param cy   center y of square
296     * @param r    radius of square
297     * @param color  fill
298     */
299    public void drawSquare(int cx, int cy, int r, Color color) {
300        // Nested loop over nearby pixels
301        for (int y = Math.max(0, cy-r); y < Math.min(image.getHeight(), cy+r); y++) {
302            for (int x = Math.max(0, cx-r); x < Math.min(image.getWidth(), cx+r); x++) {
303                image.setRGB(x, y, color.getRGB());
304            }
305        }
306    }
```

Nested loop

- **Loop from *r* rows above to *r* rows below mouse**
- **Don't go outside image boundaries (negative or > height)**
- **Loop from *r* columns left to *r* columns right of mouse**
- **Don't go outside image boundaries (negative or > width)**
- **Set pixel to color**
- **Makes a filled in square of *color***

ImageProcessor.java provides more image manipulation functionality

ImageProcessor.java

- `brighten()` does the opposite of `dim()`, but must check max color value does not go over 255
- `scaleColor()` allows each RGB component to scale individually, must cast doubles to ints with (int)
- `noise()`
  - adds random noise to each color channel
  - `Math.random()` returns number [0,1)
  - `Math.random()` * 2 -1 gives range -1..1
  - multiply that -1..1 number by scaling factor to increase range as desired

constrain() method check values to ensure they do not exceed min/max bounds

ImageProcessor.java

```
29    /**
30     * Returns a value that is one of val (if it's between min or max) or min or max
31     * @param val
32     * @param min
33     * @param max
34     * @return constrained value
35     */
36    public static double constrain(double val, double min, double max) {
37        if (val < min) {
38            return min;
39        }
40        else if (val > max) {
41            return max;
42        }
43        return val;
44    }
```

Preview: static variables or methods

- ***static*** **means variable or method is same one for all objects of this class**
- **Called "*class variable*", not instance variable**
- **Exists outside each object, accessible by all objects instantiated from the class**
- **Static variables useful when each object needs to know something about other objects (ex., how many objects have been created)**
- **Static methods useful when method does not operate on any instance variables**

Comments

- Will often need to constrain values to fall within a range (e.g., between 0 and 255 for RGB color components)
- Idea is to create a "helper" method and call it where needed, rather than duplicating range checking code in each function

Agenda

1. Image Processing Architecture
2. Manipulating individual pixels
3. Accounting for geometry
4. Puzzle

Sometimes we want a pixel value to depend on specific other pixels

ImageProcessor.java

```
155     /**
156      * Flips the current image upside down.
157      */
158     public void flip() {
159         // Create a new image into which the resulting pixels will be stored.
160         BufferedImage result = createBlankResult();
161
162         // Nested loop over every pixel
163         for (int y = 0; y < image.getHeight(); y++) { //row to pick up color
164             int y2 = image.getHeight() - 1 - y; // row to write result, note that indices go 0..height-1
165             for (int x = 0; x < image.getWidth(); x++) { //column to pick up color
166                 int color = image.getRGB(x, y); //get color from x,y
167                 result.setRGB(x, y2, color); //set color to x,y2
168             }
169         }
170
171         // Make the current image be this new image.
172         image = result;
173     }
174
```

Start with blank *result* image (*createBlankResult()* in ImageProcessor)

Flip an image upside down

Approach:

- **Create new blank image called *result***
- **Copy pixels from *image* instance variable into opposite row (e.g., row 0 copied to row *height*-1) on *result***

Sometimes we want a pixel value to depend on specific other pixels

ImageProcessor.java

```
155     /**
156      * Flips the current image upside down.
157      */
158     public void flip() {
159         // Create a new image into which the resulting pixels will be stored.
160         BufferedImage result = createBlankResult();
161
162         // Nested loop over every pixel
163         for (int y = 0; y < image.getHeight(); y++) { //row to pick up color
164             int y2 = image.getHeight() - 1 - y; // row to write result, note that indices go 0..height-1
165             for (int x = 0; x < image.getWidth(); x++) { //column to pick up color
166                 int color = image.getRGB(x, y); //get color from x,y
167                 result.setRGB(x, y2, color); //set color to x,y2
168             }
169         }
170
171         // Make the current image be this new image.
172         image = result;
173     }
174
```

Loop over all rows on *image*
This is the "source" row

Sometimes we want a pixel value to depend on specific other pixels

ImageProcessor.java

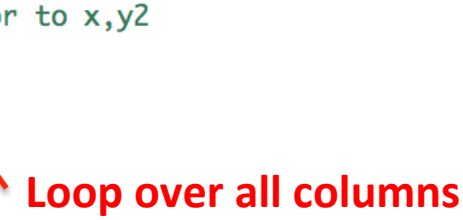
```
155     /**
156      * Flips the current image upside down.
157      */
158     public void flip() {
159         // Create a new image into which the resulting pixels will be stored.
160         BufferedImage result = createBlankResult();
161
162         // Nested loop over every pixel
163         for (int y = 0; y < image.getHeight(); y++) { //row to pick up color
164             int y2 = image.getHeight() - 1 - y; // row to write result, note that indices go 0..height-1
165             for (int x = 0; x < image.getWidth(); x++) { //column to pick up color
166                 int color = image.getRGB(x, y); //get color from x,y
167                 result.setRGB(x, y2, color); //set color to x,y2
168             }
169         }
170
171         // Make the current image be this new image.
172         image = result;
173     }
174
```

Calculate "target" row on opposite end of image (e.g., row 0 goes to row height-1)

Sometimes we want a pixel value to depend on specific other pixels

ImageProcessor.java

```
155     /**
156      * Flips the current image upside down.
157      */
158     public void flip() {
159         // Create a new image into which the resulting pixels will be stored.
160         BufferedImage result = createBlankResult();
161
162         // Nested loop over every pixel
163         for (int y = 0; y < image.getHeight(); y++) { //row to pick up color
164             int y2 = image.getHeight() - 1 - y; // row to write result, note that indices go 0..height-1
165             for (int x = 0; x < image.getWidth(); x++) { //column to pick up color
166                 int color = image.getRGB(x, y); //get color from x,y
167                 result.setRGB(x, y2, color); //set color to x,y2
168             }
169         }
170
171         // Make the current image be this new image.
172         image = result;
173     }
174
```



Sometimes we want a pixel value to depend on specific other pixels

ImageProcessor.java

```
155     /**
156      * Flips the current image upside down.
157      */
158     public void flip() {
159         // Create a new image into which the resulting pixels will be stored.
160         BufferedImage result = createBlankResult();
161
162         // Nested loop over every pixel
163         for (int y = 0; y < image.getHeight(); y++) { //row to pick up color
164             int y2 = image.getHeight() - 1 - y; // row to write result, note that indices go 0..height-1
165             for (int x = 0; x < image.getWidth(); x++) { //column to pick up color
166                 int color = image.getRGB(x, y); //get color from x,y
167                 result.setRGB(x, y2, color); //set color to x,y2
168             }
169         }
170
171         // Make the current image be this new image.
172         image = result;
173     }
174
```

Pick up color at "source location" x,y on *image* instance variable

Write color to "target location" x,y2 on *result* image

Sometimes we want a pixel value to depend on specific other pixels

ImageProcessor.java

```
155     /**
156      * Flips the current image upside down.
157      */
158     public void flip() {
159         // Create a new image into which the resulting pixels will be stored.
160         BufferedImage result = createBlankResult();
161
162         // Nested loop over every pixel
163         for (int y = 0; y < image.getHeight(); y++) { //row to pick up color
164             int y2 = image.getHeight() - 1 - y; // row to write result, note that indices go 0..height-1
165             for (int x = 0; x < image.getWidth(); x++) { //column to pick up color
166                 int color = image.getRGB(x, y); //get color from x,y
167                 result.setRGB(x, y2, color); //set color to x,y2
168             }
169         }
170
171         // Make the current image be this new image.
172         image = result;
173     }
174
```

Set *image* instance variable to *result*

What happens if we do not run line 172?
Loose changes! (*result* is local variable)

Original *image* will be garbage collected

Sometimes we want a pixel value to depend on specific other pixels

ImageProcessor.java

```
155     /**
156      * Flips the current image upside down.
157      */
158     public void flip() {
159         // Create a new image into which the resulting pixels will be stored.
160         BufferedImage result = createBlankResult();
161
162         // Nested loop over every pixel
163         for (int y = 0; y < image.getHeight(); y++) { //row to pick up color
164             int y2 = image.getHeight() - 1 - y; // row to write result, note that indices go 0..height-1
165             for (int x = 0; x < image.getWidth(); x++) { //column to pick up color
166                 int color = image.getRGB(x, y); //get color from x,y
167                 result.setRGB(x, y2, color); //set color to x,y2
168             }
169         }
170
171         // Make the current image be this new image.
172         image = result;
173     }
174
```

What would happen if we had not started with a blank image, but instead tried to update pixel colors in place?

Can you think of way to avoid using a second, initially blank image?

Sometimes we want a pixel value to depend on specific other pixels

ImageProcessor.java

```
155     /**
156      * Flips the current image upside down.
157      */
158     public void flip() {
159         // Create a new image into which the resulting pixels will be stored.
160         BufferedImage result = createBlankResult();
161
162         // Nested loop over every pixel
163         for (int y = 0; y < image.getHeight(); y++) { //row to pick up color
164             int y2 = image.getHeight() - 1 - y; // row to write result, note that indices go 0..height-1
165             for (int x = 0; x < image.getWidth(); x++) { //column to pick up color
166                 int color = image.getRGB(x, y); //get color from x,y
167                 result.setRGB(x, y2, color); //set color to x,y2
168             }
169         }
170
171         // Make the current image be this new image.
172         image = result;
173     }
174
```

Classic swap pattern

To swap variables a and b:

- **temp = a**
- **a = b**
- **b = temp**

To solve without second image:

temp = getRGB(x, y) //temp = a

setRGB(x, y, getRGB(x,y2)) //a = b

setRGB(x, y2, temp) //b = temp

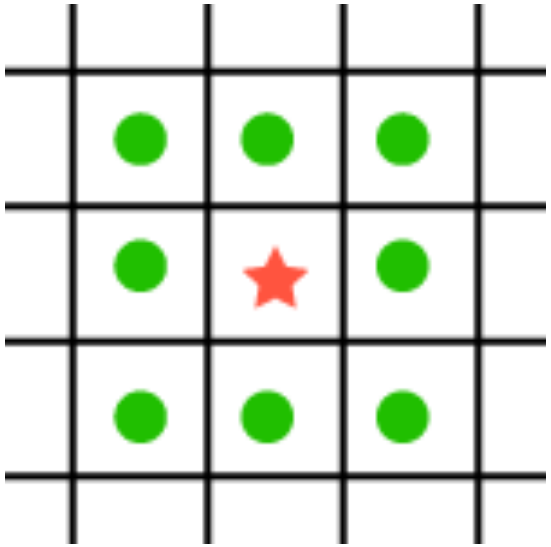
What would happen if we had not started with a blank image, but instead tried to update pixel colors in place?

Can you think of way to avoid using a second, initially blank image?

Sometimes we want to operate on a pixel's neighbors

Blur image by averaging around each pixel's neighbors

Pixel and neighbors



Averaging can smooth outliers

10	12	13
12	34	11
10	13	11

Average = (10+12+13+12+34+11 +10+13+11)/9 = 14

Replace all values in new image with average of all neighbors

average() examines neighbors to smooth (blur) an image

ImageProcessor.java

```
66    public void average(int radius) {
67        // Create a new image into which the resulting pixels will be stored.
68        BufferedImage result = createBlankResult();
69
70        // Nested loop over every pixel
71        for (int y = 0; y < image.getHeight(); y++) {
72            for (int x = 0; x < image.getWidth(); x++) {
73                int sumR = 0, sumG = 0, sumB = 0;
74                int n = 0;
75                // Nested loop over neighbors
76                // but be careful not to go outside image (max, min stuff).
77                for (int ny = Math.max(0, y - radius);
78                        ny < Math.min(image.getHeight(), y + 1 + radius);
79                        ny++) {
80                    for (int nx = Math.max(0, x - radius);
81                            nx < Math.min(image.getWidth(), x + 1 + radius);
82                            nx++) {
83                        // Add all the neighbors (& self) to the running totals
84                        Color c = new Color(image.getRGB(nx, ny));
85                        sumR += c.getRed();
86                        sumG += c.getGreen();
87                        sumB += c.getBlue();
88                        n++;
89                    }
90                }
91                Color newColor = new Color(sumR/n, sumG/n, sumB/n);
92                result.setRGB(x, y, newColor.getRGB());
93            }
94        }
95
96        // Make the current image be this new image.
97        image = result;
98    }
```

Create blank *result*

Nested loop to cover all x,y locations

average() examines neighbors to smooth (blur) an image

ImageProcessor.java

```
66    public void average(int radius) {
67        // Create a new image into which the resulting pixels will be stored.
68        BufferedImage result = createBlankResult();
69
70        // Nested loop over every pixel
71        for (int y = 0; y < image.getHeight(); y++) {
72            for (int x = 0; x < image.getWidth(); x++) {
73                int sumR = 0, sumG = 0, sumB = 0;
74                int n = 0;
75                // Nested loop over neighbors
76                // but be careful not to go outside image (max, min stuff).
77                for (int ny = Math.max(0, y - radius);
78                        ny < Math.min(image.getHeight(), y + 1 + radius);
79                        ny++) {
80                    for (int nx = Math.max(0, x - radius);
81                            nx < Math.min(image.getWidth(), x + 1 + radius);
82                            nx++) {
83                        // Add all the neighbors (& self) to the running totals
84                        Color c = new Color(image.getRGB(nx, ny));
85                        sumR += c.getRed();
86                        sumG += c.getGreen();
87                        sumB += c.getBlue();
88                        n++;
89                    }
90                }
91                Color newColor = new Color(sumR/n, sumG/n, sumB/n);
92                result.setRGB(x, y, newColor.getRGB());
93            }
94        }
95
96        // Make the current image be this new image.
97        image = result;
98    }
```

Will calculate average R,G,B values of surrounding pixels

average() examines neighbors to smooth (blur) an image

ImageProcessor.java

```
66      public void average(int radius) {
67          // Create a new image into which the resulting pixels will be stored.
68          BufferedImage result = createBlankResult();
69
70          // Nested loop over every pixel
71          for (int y = 0; y < image.getHeight(); y++) {
72              for (int x = 0; x < image.getWidth(); x++) {
73                  int sumR = 0, sumG = 0, sumB = 0;
74                  int n = 0;
75                  // Nested loop over neighbors
76                  // but be careful not to go outside image (max, min stuff).
77                  for (int ny = Math.max(0, y - radius);
78                          ny < Math.min(image.getHeight(), y + 1 + radius);
79                          ny++) {
80                      for (int nx = Math.max(0, x - radius);
81                              nx < Math.min(image.getWidth(), x + 1 + radius);
82                              nx++) {
83                          // Add all the neighbors (& self) to the running totals
84                          Color c = new Color(image.getRGB(nx, ny));
85                          sumR += c.getRed();
86                          sumG += c.getGreen();
87                          sumB += c.getBlue();
88                          n++;
89                      }
90                  }
91                  Color newColor = new Color(sumR/n, sumG/n, sumB/n);
92                  result.setRGB(x, y, newColor.getRGB());
93              }
94          }
95
96          // Make the current image be this new image.
97          image = result;
98      }
```

For each x,y location get neighbors:

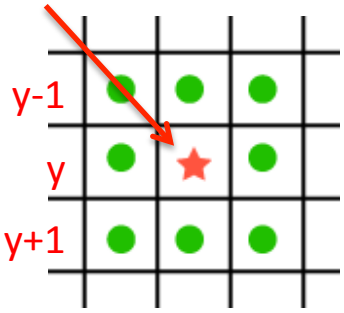
- **Loop from *radius* rows above to *radius* rows below current row**
- **Be careful not to have a negative row index or an index ≥ max width/height**

average() examines neighbors to smooth (blur) an image

ImageProcessor.java

```
66      public void average(int radius) {
67          // Create a new image into which the resulting pixels will be stored.
68          BufferedImage result = createBlankResult();
69
70          // Nested loop over every pixel
71          for (int y = 0; y < image.getHeight(); y++) {
72              for (int x = 0; x < image.getWidth(); x++) {
73                  int sumR = 0, sumG = 0, sumB = 0;
74                  int n = 0;
75                  // Nested loop over neighbors
76                  // but be careful not to go outside image (max, min stuff).
77                  for (int ny = Math.max(0, y - radius);
78                          ny < Math.min(image.getHeight(), y + 1 + radius);
79                          ny++) {
80                      for (int nx = Math.max(0, x - radius);
81                              nx < Math.min(image.getWidth(), x + 1 + radius);
82                              nx++) {
83                          // Add all the neighbors (& self) to the running totals
84                          Color c = new Color(image.getRGB(nx, ny));
85                          sumR += c.getRed();
86                          sumG += c.getGreen();
87                          sumB += c.getBlue();
88                          n++;
89                      }
90                  }
91                  Color newColor = new Color(sumR/n, sumG/n, sumB/n);
92                  result.setRGB(x, y, newColor.getRGB());
93              }
94          }
95
96          // Make the current image be this new image.
97          image = result;
98      }
```

Pixel at x,y



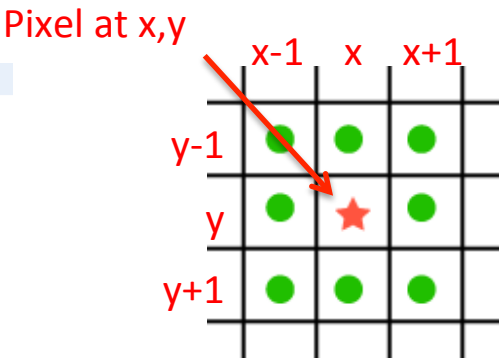
For each x,y location get neighbors:

- **Loop from *radius* rows above to *radius* rows below current row**
- **Be careful not to have a negative row index**

average() examines neighbors to smooth (blur) an image

ImageProcessor.java

```
66    public void average(int radius) {
67        // Create a new image into which the resulting pixels will be stored.
68        BufferedImage result = createBlankResult();
69
70        // Nested loop over every pixel
71        for (int y = 0; y < image.getHeight(); y++) {
72            for (int x = 0; x < image.getWidth(); x++) {
73                int sumR = 0, sumG = 0, sumB = 0;
74                int n = 0;
75                // Nested loop over neighbors
76                // but be careful not to go outside image (max, min stuff).
77                for (int ny = Math.max(0, y - radius);
78                        ny < Math.min(image.getHeight(), y + 1 + radius);
79                        ny++) {
80                    for (int nx = Math.max(0, x - radius);
81                            nx < Math.min(image.getWidth(), x + 1 + radius);
82                            nx++) {
83                        // Add all the neighbors (& self) to the running totals
84                        Color c = new Color(image.getRGB(nx, ny));
85                        sumR += c.getRed();
86                        sumG += c.getGreen();
87                        sumB += c.getBlue();
88                        n++;
89                    }
90                }
91                Color newColor = new Color(sumR/n, sumG/n, sumB/n);
92                result.setRGB(x, y, newColor.getRGB());
93            }
94        }
95
96        // Make the current image be this new image.
97        image = result;
98    }
```



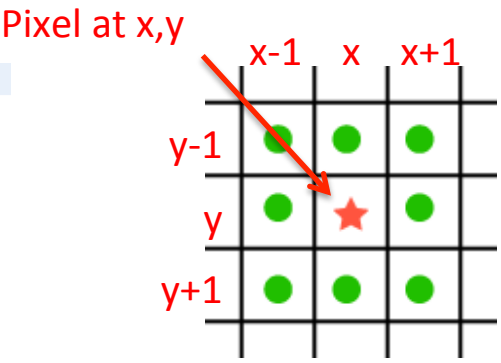
For each x,y location get neighbors:

- **Loop from *radius* columns left to *radius* columns right of current column**
- **Be careful not to have a negative column index**

average() examines neighbors to smooth (blur) an image

ImageProcessor.java

```
66      public void average(int radius) {
67          // Create a new image into which the resulting pixels will be stored.
68          BufferedImage result = createBlankResult();
69
70          // Nested loop over every pixel
71          for (int y = 0; y < image.getHeight(); y++) {
72              for (int x = 0; x < image.getWidth(); x++) {
73                  int sumR = 0, sumG = 0, sumB = 0;
74                  int n = 0;
75                  // Nested loop over neighbors
76                  // but be careful not to go outside image (max, min stuff).
77                  for (int ny = Math.max(0, y - radius);
78                          ny < Math.min(image.getHeight(), y + 1 + radius);
79                          ny++) {
80                      for (int nx = Math.max(0, x - radius);
81                              nx < Math.min(image.getWidth(), x + 1 + radius);
82                              nx++) {
83                          // Add all the neighbors (& self) to the running totals
84                          Color c = new Color(image.getRGB(nx, ny));
85                          sumR += c.getRed();
86                          sumG += c.getGreen();
87                          sumB += c.getBlue();
88                          n++;
89                      }
90                  }
91                  Color newColor = new Color(sumR/n, sumG/n, sumB/n);
92                  result.setRGB(x, y, newColor.getRGB());
93              }
94          }
95
96          // Make the current image be this new image.
97          image = result;
98      }
```

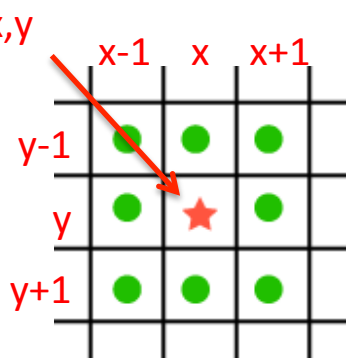


Why is this y +1?
And this x+1?

average() examines neighbors to smooth (blur) an image

ImageProcessor.java

```
66    public void average(int radius) {
67        // Create a new image into which the resulting pixels will be stored.
68        BufferedImage result = createBlankResult();
69
70        // Nested loop over every pixel
71        for (int y = 0; y < image.getHeight(); y++) {
72            for (int x = 0; x < image.getWidth(); x++) {
73                int sumR = 0, sumG = 0, sumB = 0;
74                int n = 0;
75                // Nested loop over neighbors
76                // but be careful not to go outside image (max, min stuff).
77                for (int ny = Math.max(0, y - radius);
78                        ny < Math.min(image.getHeight(), y + 1 + radius);
79                        ny++) {
80                    for (int nx = Math.max(0, x - radius);
81                            nx < Math.min(image.getWidth(), x + 1 + radius);
82                            nx++) {
83                        // Add all the neighbors (& self) to the running totals
84                        Color c = new Color(image.getRGB(nx, ny));
85                        sumR += c.getRed();
86                        sumG += c.getGreen();
87                        sumB += c.getBlue();
88                        n++;
89                    }
90                }
91                Color newColor = new Color(sumR/n, sumG/n, sumB/n);
92                result.setRGB(x, y, newColor.getRGB());
93            }
94        }
95
96        // Make the current image be this new image.
97        image = result;
98    }
```



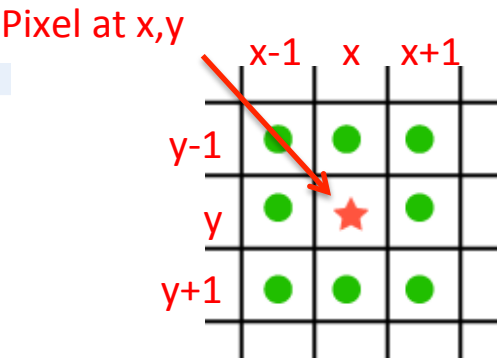
Why is this y +1?
And this x+1?

Because the check in the for loop is < not ≤

average() examines neighbors to smooth (blur) an image

ImageProcessor.java

```
66      public void average(int radius) {
67          // Create a new image into which the resulting pixels will be stored.
68          BufferedImage result = createBlankResult();
69
70          // Nested loop over every pixel
71          for (int y = 0; y < image.getHeight(); y++) {
72              for (int x = 0; x < image.getWidth(); x++) {
73                  int sumR = 0, sumG = 0, sumB = 0;
74                  int n = 0;
75                  // Nested loop over neighbors
76                  // but be careful not to go outside image (max, min stuff).
77                  for (int ny = Math.max(0, y - radius);
78                          ny < Math.min(image.getHeight(), y + 1 + radius);
79                          ny++) {
80                      for (int nx = Math.max(0, x - radius);
81                              nx < Math.min(image.getWidth(), x + 1 + radius);
82                              nx++) {
83                          // Add all the neighbors (& self) to the running totals
84                          Color c = new Color(image.getRGB(nx, ny));
85                          sumR += c.getRed();
86                          sumG += c.getGreen();
87                          sumB += c.getBlue();
88                          n++;
89                      }
90                  }
91                  Color newColor = new Color(sumR/n, sumG/n, sumB/n);
92                  result.setRGB(x, y, newColor.getRGB());
93              }
94          }
95
96          // Make the current image be this new image.
97          image = result;
98      }
```



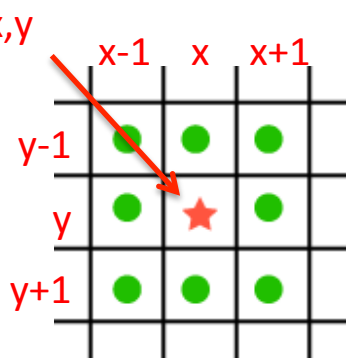
Pick up color components for this neighboring pixel and sum them

Also keep track of how many neighbors pixel has in variable *n* (might not be 9)

average() examines neighbors to smooth (blur) an image

ImageProcessor.java

```
66      public void average(int radius) {
67          // Create a new image into which the resulting pixels will be stored.
68          BufferedImage result = createBlankResult();
69
70          // Nested loop over every pixel
71          for (int y = 0; y < image.getHeight(); y++) {
72              for (int x = 0; x < image.getWidth(); x++) {
73                  int sumR = 0, sumG = 0, sumB = 0;
74                  int n = 0;
75                  // Nested loop over neighbors
76                  // but be careful not to go outside image (max, min stuff).
77                  for (int ny = Math.max(0, y - radius);
78                          ny < Math.min(image.getHeight(), y + 1 + radius);
79                          ny++) {
80                      for (int nx = Math.max(0, x - radius);
81                              nx < Math.min(image.getWidth(), x + 1 + radius);
82                              nx++) {
83                          // Add all the neighbors (& self) to the running totals
84                          Color c = new Color(image.getRGB(nx, ny));
85                          sumR += c.getRed();
86                          sumG += c.getGreen();
87                          sumB += c.getBlue();
88                          n++;
89                      }
90                  }
91                  Color newColor = new Color(sumR/n, sumG/n, sumB/n);
92                  result.setRGB(x, y, newColor.getRGB());
93              }
94          }
95
96          // Make the current image be this new image.
97          image = result;
98      }
```

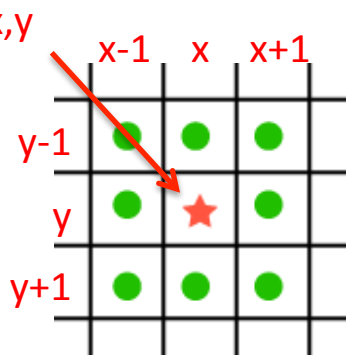


Calculate average and write it to result

average() examines neighbors to smooth (blur) an image

ImageProcessor.java

```
66      public void average(int radius) {
67          // Create a new image into which the resulting pixels will be stored.
68          BufferedImage result = createBlankResult();
69
70          // Nested loop over every pixel
71          for (int y = 0; y < image.getHeight(); y++) {
72              for (int x = 0; x < image.getWidth(); x++) {
73                  int sumR = 0, sumG = 0, sumB = 0;
74                  int n = 0;
75                  // Nested loop over neighbors
76                  // but be careful not to go outside image (max, min stuff).
77                  for (int ny = Math.max(0, y - radius);
78                          ny < Math.min(image.getHeight(), y + 1 + radius);
79                          ny++) {
80                      for (int nx = Math.max(0, x - radius);
81                              nx < Math.min(image.getWidth(), x + 1 + radius);
82                              nx++) {
83                          // Add all the neighbors (& self) to the running totals
84                          Color c = new Color(image.getRGB(nx, ny));
85                          sumR += c.getRed();
86                          sumG += c.getGreen();
87                          sumB += c.getBlue();
88                          n++;
89                      }
90                  }
91                  Color newColor = new Color(sumR/n, sumG/n, sumB/n);
92                  result.setRGB(x, y, newColor.getRGB());
93              }
94          }
95
96          // Make the current image be this new image.
97          image = result;
98      }
```

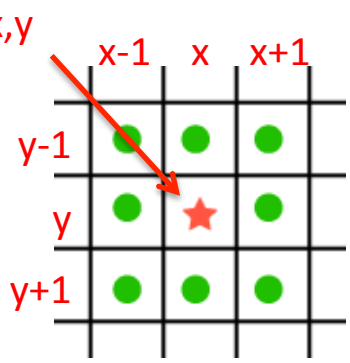


Are we done?

average() examines neighbors to smooth (blur) an image

ImageProcessor.java

```
66      public void average(int radius) {
67          // Create a new image into which the resulting pixels will be stored.
68          BufferedImage result = createBlankResult();
69
70          // Nested loop over every pixel
71          for (int y = 0; y < image.getHeight(); y++) {
72              for (int x = 0; x < image.getWidth(); x++) {
73                  int sumR = 0, sumG = 0, sumB = 0;
74                  int n = 0;
75                  // Nested loop over neighbors
76                  // but be careful not to go outside image (max, min stuff).
77                  for (int ny = Math.max(0, y - radius);
78                          ny < Math.min(image.getHeight(), y + 1 + radius);
79                          ny++) {
80                      for (int nx = Math.max(0, x - radius);
81                              nx < Math.min(image.getWidth(), x + 1 + radius);
82                              nx++) {
83                          // Add all the neighbors (& self) to the running totals
84                          Color c = new Color(image.getRGB(nx, ny));
85                          sumR += c.getRed();
86                          sumG += c.getGreen();
87                          sumB += c.getBlue();
88                          n++;
89                      }
90                  }
91                  Color newColor = new Color(sumR/n, sumG/n, sumB/n);
92                  result.setRGB(x, y, newColor.getRGB());
93              }
94          }
95
96          // Make the current image be this new image.
97          image = result;
98      }
```



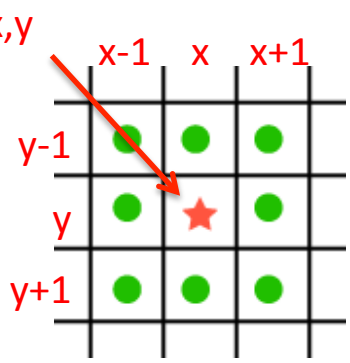
Are we done? NO, must set *image=result* to save change

Otherwise changes would be lost! Why?

average() examines neighbors to smooth (blur) an image

ImageProcessor.java

```
66      public void average(int radius) {
67          // Create a new image into which the resulting pixels will be stored.
68          BufferedImage result = createBlankResult();
69
70          // Nested loop over every pixel
71          for (int y = 0; y < image.getHeight(); y++) {
72              for (int x = 0; x < image.getWidth(); x++) {
73                  int sumR = 0, sumG = 0, sumB = 0;
74                  int n = 0;
75                  // Nested loop over neighbors
76                  // but be careful not to go outside image (max, min stuff).
77                  for (int ny = Math.max(0, y - radius);
78                          ny < Math.min(image.getHeight(), y + 1 + radius);
79                          ny++) {
80                      for (int nx = Math.max(0, x - radius);
81                              nx < Math.min(image.getWidth(), x + 1 + radius);
82                              nx++) {
83                          // Add all the neighbors (& self) to the running totals
84                          Color c = new Color(image.getRGB(nx, ny));
85                          sumR += c.getRed();
86                          sumG += c.getGreen();
87                          sumB += c.getBlue();
88                          n++;
89                      }
90                  }
91                  Color newColor = new Color(sumR/n, sumG/n, sumB/n);
92                  result.setRGB(x, y, newColor.getRGB());
93              }
94          }
95
96          // Make the current image be this new image.
97          image = result;
98      }
```



Are we done? NO, must set *image=result* to save change

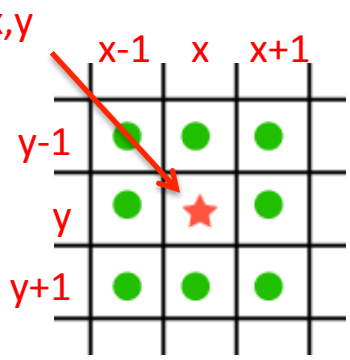
Otherwise changes would be lost! Why?

***result* is a local variable, goes out of scope**

average() examines neighbors to smooth (blur) an image

ImageProcessor.java

```
66      public void average(int radius) {
67          // Create a new image into which the resulting pixels will be stored.
68          BufferedImage result = createBlankResult();
69
70          // Nested loop over every pixel
71          for (int y = 0; y < image.getHeight(); y++) {
72              for (int x = 0; x < image.getWidth(); x++) {
73                  int sumR = 0, sumG = 0, sumB = 0;
74                  int n = 0;
75                  // Nested loop over neighbors
76                  // but be careful not to go outside image (max, min stuff).
77                  for (int ny = Math.max(0, y - radius);
78                          ny < Math.min(image.getHeight(), y + 1 + radius);
79                          ny++) {
80                      for (int nx = Math.max(0, x - radius);
81                              nx < Math.min(image.getWidth(), x + 1 + radius);
82                              nx++) {
83                          // Add all the neighbors (& self) to the running totals
84                          Color c = new Color(image.getRGB(nx, ny));
85                          sumR += c.getRed();
86                          sumG += c.getGreen();
87                          sumB += c.getBlue();
88                          n++;
89                      }
90                  }
91                  Color newColor = new Color(sumR/n, sumG/n, sumB/n);
92                  result.setRGB(x, y, newColor.getRGB());
93              }
94          }
95
96          // Make the current image be this new image.
97          image = result;
98      }
```

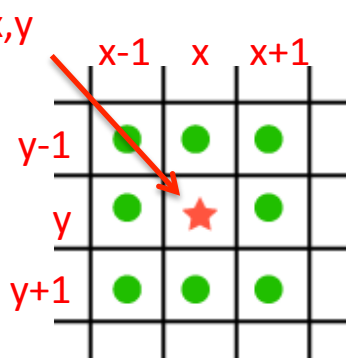


Could we do this in place, just making changes on the original image instead of using a second image?

average() examines neighbors to smooth (blur) an image

ImageProcessor.java

```
66      public void average(int radius) {
67          // Create a new image into which the resulting pixels will be stored.
68          BufferedImage result = createBlankResult();
69
70          // Nested loop over every pixel
71          for (int y = 0; y < image.getHeight(); y++) {
72              for (int x = 0; x < image.getWidth(); x++) {
73                  int sumR = 0, sumG = 0, sumB = 0;
74                  int n = 0;
75                  // Nested loop over neighbors
76                  // but be careful not to go outside image (max, min stuff).
77                  for (int ny = Math.max(0, y - radius);
78                          ny < Math.min(image.getHeight(), y + 1 + radius);
79                          ny++) {
80                      for (int nx = Math.max(0, x - radius);
81                              nx < Math.min(image.getWidth(), x + 1 + radius);
82                              nx++) {
83                          // Add all the neighbors (& self) to the running totals
84                          Color c = new Color(image.getRGB(nx, ny));
85                          sumR += c.getRed();
86                          sumG += c.getGreen();
87                          sumB += c.getBlue();
88                          n++;
89                      }
90                  }
91                  Color newColor = new Color(sumR/n, sumG/n, sumB/n);
92                  result.setRGB(x, y, newColor.getRGB());
93              }
94          }
95
96          // Make the current image be this new image.
97          image = result;
98      }
```



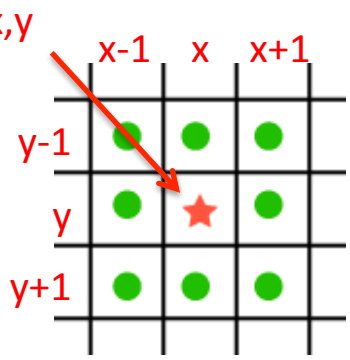
Could we do this in place, just making changes on the original image instead of using a second image?

NO, the changes would affect the average value of each pixel

average() examines neighbors to smooth (blur) an image

ImageProcessor.java

```
66      public void average(int radius) {
67          // Create a new image into which the resulting pixels will be stored.
68          BufferedImage result = createBlankResult();
69
70          // Nested loop over every pixel
71          for (int y = 0; y < image.getHeight(); y++) {
72              for (int x = 0; x < image.getWidth(); x++) {
73                  int sumR = 0, sumG = 0, sumB = 0;
74                  int n = 0;
75                  // Nested loop over neighbors
76                  // but be careful not to go outside image (max, min stuff).
77                  for (int ny = Math.max(0, y - radius);
78                          ny < Math.min(image.getHeight(), y + 1 + radius);
79                          ny++) {
80                      for (int nx = Math.max(0, x - radius);
81                              nx < Math.min(image.getWidth(), x + 1 + radius);
82                              nx++) {
83                          // Add all the neighbors (& self) to the running totals
84                          Color c = new Color(image.getRGB(nx, ny));
85                          sumR += c.getRed();
86                          sumG += c.getGreen();
87                          sumB += c.getBlue();
88                          n++;
89                      }
90                  }
91                  Color newColor = new Color(sumR/n, sumG/n, sumB/n);
92                  result.setRGB(x, y, newColor.getRGB());
93              }
94          }
95
96          // Make the current image be this new image.
97          image = result;
98      }
```

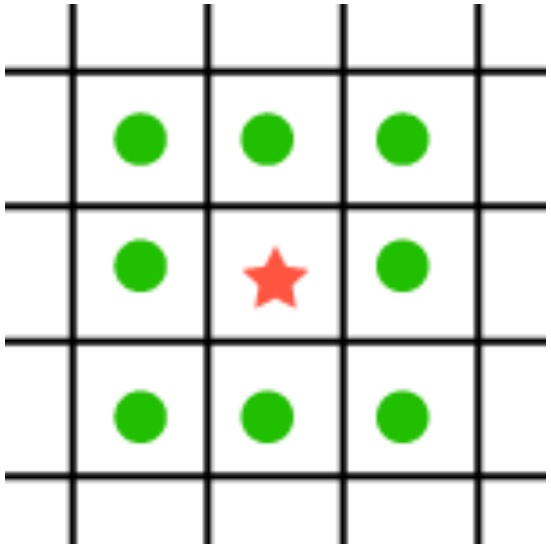


CAUTION: this algorithm is very inefficient, if you set *radius* to a large number, it will take a *LONG* time to finish

sharpen() works similarly to *average()*, but subtracts neighbors weights

Sharpen image by subtracting each pixel's neighbors

Pixel and neighbors



Subtract neighbor weights

-1	-1	-1
-1	9	-1
-1	-1	-1

Result= pixel * 9 – sum(neighbors)

- **Replace all values in new image with computed value**
- **This is called convolution**
- **Used in deep learning and signal processing**

Adding some interactivity by handling key and mouse presses

ImageProcessorGUI.handleKeyPress()

- Get key pressed
- Call appropriate function on ImageProcessor
- *repaint() at end*

To get image from slide 3 press:

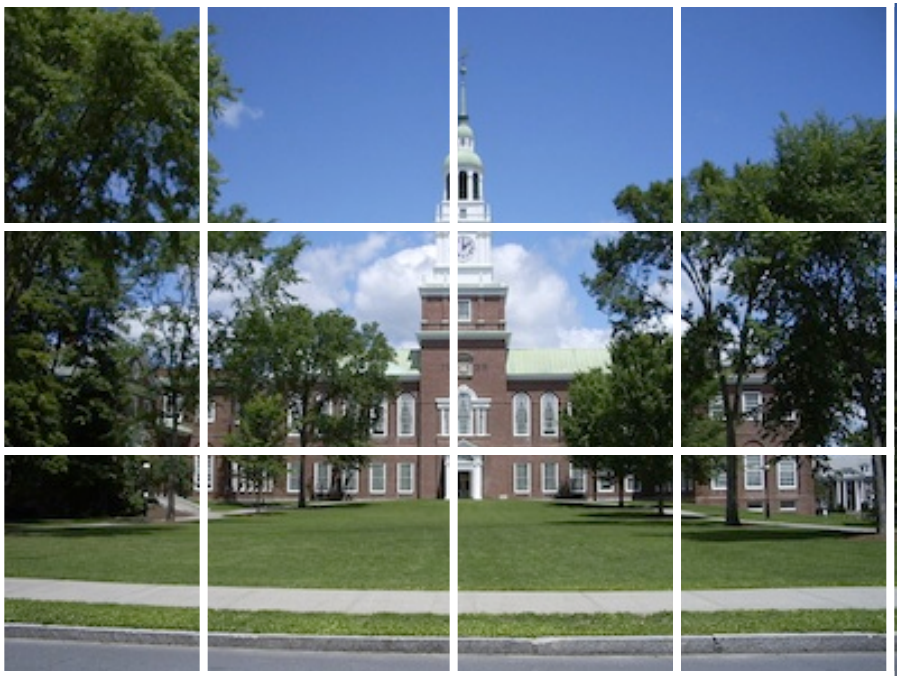
- 'g' to convert to gray scale (see *gray()* in ImageProcessor)
- 'h' for sharpen

Agenda

1. Image Processing Architecture
2. Manipulating individual pixels
3. Accounting for geometry
4. Puzzle

Puzzle breaks an image into multiple pieces and stores pieces in an ArrayList

Original image



3 x 4 puzzle pieces

Piece 0	Piece 1	Piece 2	Piece 3
Piece 4	Piece 5	Piece 6	Piece 7
Piece 8	Piece 9	Piece 10	Piece 11

Each piece will be stored in an ArrayList

PuzzleGUI.java Demo

- Image of Baker broken into pieces and shuffled
- Click on two pieces to swap
- Reconstruct image of Baker

Puzzle.java: Breaks original image into pieces and shuffles them

ArrayList to hold BufferedImage pieces (portions of original image)

```
11 public class Puzzle {
12     private BufferedImage image;                    // the whole original
13     private int numRows, numCols;                    // how many piece
14     private int pieceWidth, pieceHeight;             // size of pieces
15     private ArrayList<BufferedImage> pieces;         // the small images
16
17     public Puzzle(BufferedImage image, int numRows, int numCols) {
18         this.image = image;
19         this.numRows = numRows;
20         this.numCols = numCols;
21
22         // Make and shuffle the pieces.
23         createPieces();
24         shufflePieces();
25     }
26
```

Puzzle.java: Breaks original image into pieces and shuffles them

```
11 public class Puzzle {
12     private BufferedImage image;                    // the whole original
13     private int numRows, numCols;                    // how many piece
14     private int pieceWidth, pieceHeight;             // size of pieces
15     private ArrayList<BufferedImage> pieces;         // the small images
16
17     public Puzzle(BufferedImage image, int numRows, int numCols) {
18         this.image = image;
19         this.numRows = numRows;
20         this.numCols = numCols;
21
22         // Make and shuffle the pieces.
23         createPieces();
24         shufflePieces();
25     }
26
```

ArrayList to hold BufferedImage pieces (portions of original image)

Save image and number of rows and columns into which image will be split (e.g., 3 rows x 4 columns will make 12 pieces)

Puzzle.java: Breaks original image into pieces and shuffles them

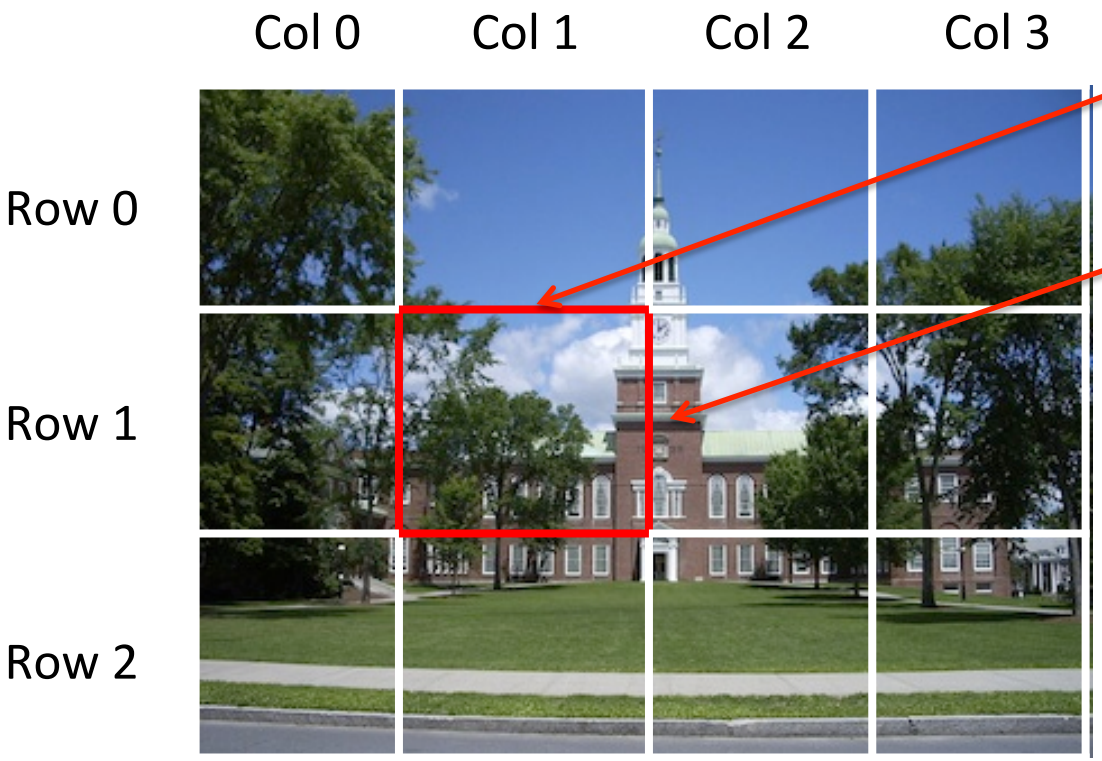
```
11 public class Puzzle {
12     private BufferedImage image;                    // the whole original
13     private int numRows, numCols;                    // how many piece
14     private int pieceWidth, pieceHeight;             // size of pieces
15     private ArrayList<BufferedImage> pieces;         // the small images
16
17     public Puzzle(BufferedImage image, int numRows, int numCols) {
18         this.image = image;
19         this.numRows = numRows;
20         this.numCols = numCols;
21
22         // Make and shuffle the pieces.
23         createPieces();
24         shufflePieces();
25     }
26
```

ArrayList to hold pieces (portions of original image)

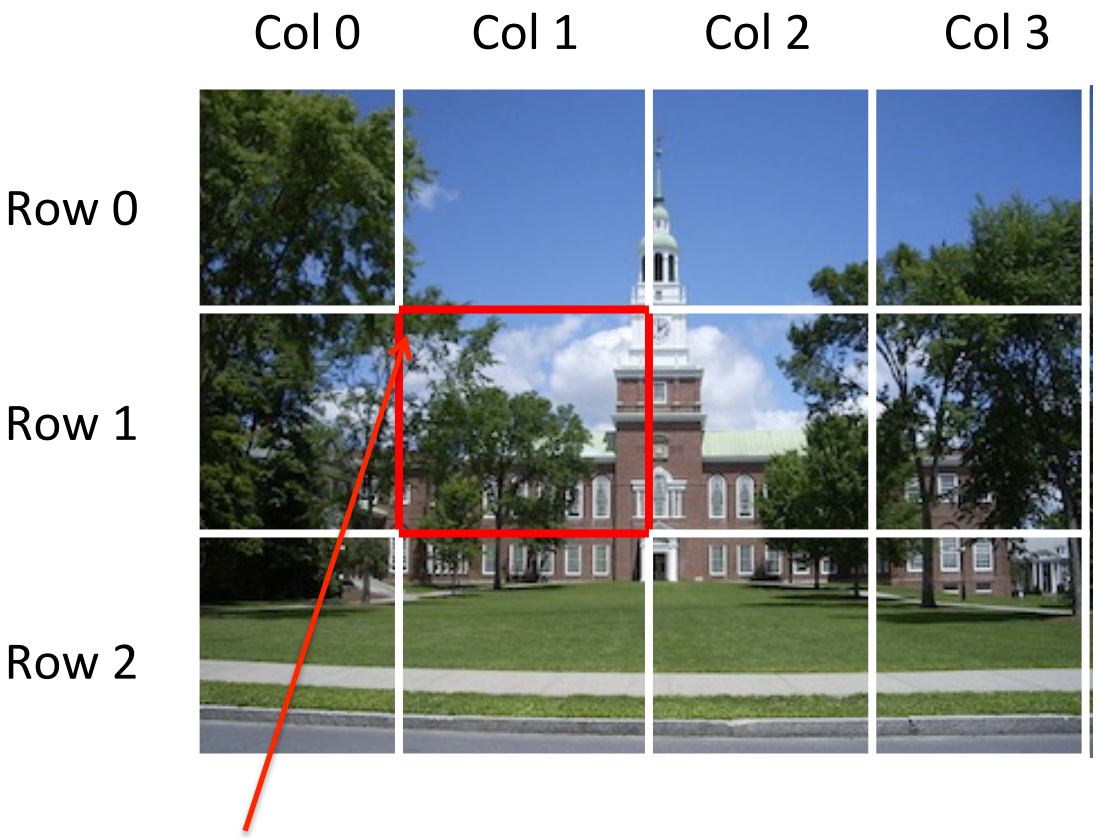
Save image and number of rows and columns into which image will be split (e.g., 3 rows x 4 columns will make 12 pieces)

Split image in 3 x 4 pieces and then shuffle them

Puzzle.java: createPieces() breaks image into separate pieces



Puzzle.java: createPieces() breaks image into separate pieces

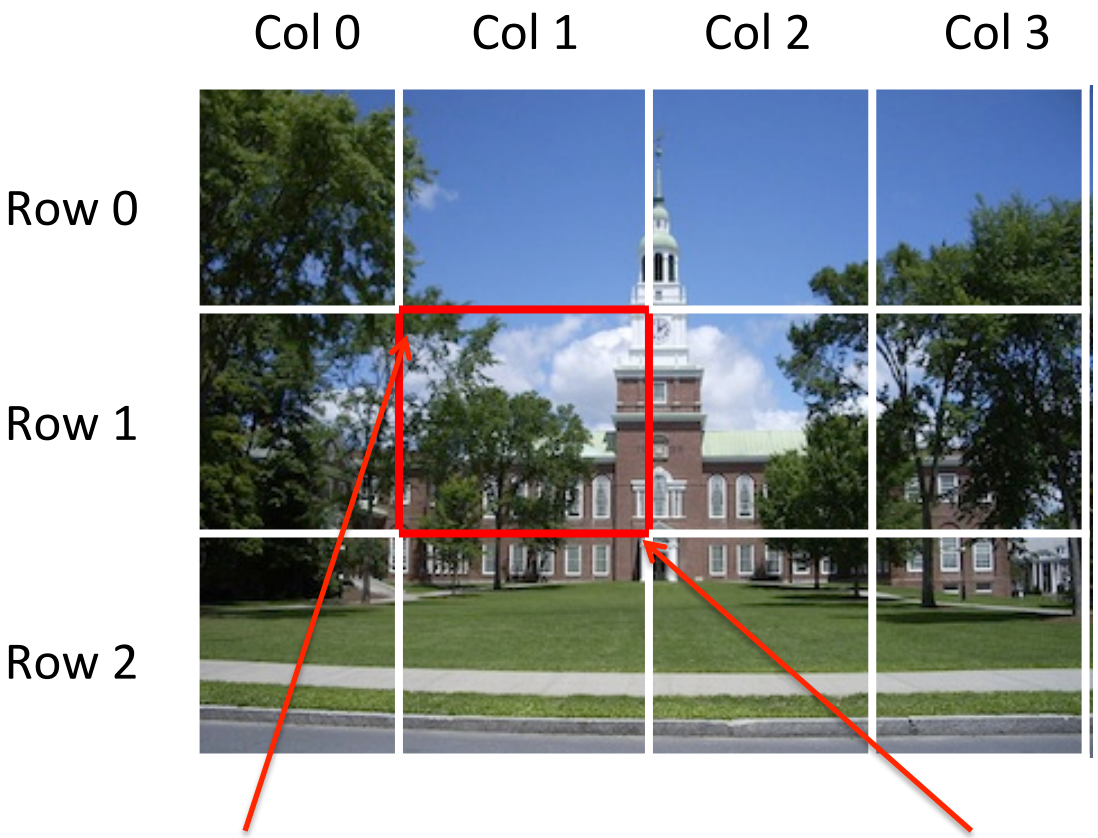


pieceWidth = image.getWidth()/numCols

pieceHeight = image.getHeight()/numRows

Top left coordinates:
x = col * pieceWidth
y = row * pieceHeight

Puzzle.java: createPieces() breaks image into separate pieces



pieceWidth = image.getWidth()/numCols

pieceHeight = image.getHeight()/numRows

Top left coordinates:
x = col * pieceWidth
y = row * pieceHeight

Bottom right coordinates:
x1 = x + pieceWidth
y1 = y + pieceHeight

Puzzle.java: createPieces() breaks image into separate pieces

Calculate width and height of each piece, save in instance variables

```
50  /**
51   * Creates the pieces list, fragments from the image.
52   */
53  private void createPieces() {
54      // Compute piece size according to how many have to fit in im
55      pieceWidth = image.getWidth() / numCols;
56      pieceHeight = image.getHeight() / numRows;
57      // Create and fill up the array, iterating by piece row and p
58      pieces = new ArrayList<BufferedImage>();
59      for (int r = 0; r < numRows; r++) {
60          for (int c = 0; c < numCols; c++) {
61              pieces.add(getSubimage(image, c*pieceWidth, r*pieceHe
62          }
63      }
64  }
65
```

Puzzle.java: createPieces() breaks image into separate pieces

```
50  /**
51   * Creates the pieces list, fragments from the image.
52   */
53  private void createPieces() {
54      // Compute piece size according to how many have to fit in im
55      pieceWidth = image.getWidth() / numCols;
56      pieceHeight = image.getHeight() / numRows;
57      // Create and fill up the array, iterating by piece row and p
58      pieces = new ArrayList<BufferedImage>();
59      for (int r = 0; r < numRows; r++) {
60          for (int c = 0; c < numCols; c++) {
61              pieces.add(getSubimage(image, c*pieceWidth, r*pieceHe
62          }
63      }
64  }
65
```

Calculate width and height of each piece, save in instance variables

For each row and column (nested loop)

- **Grab a portion (piece) of the original image**
- ***getSubImage()* copies pixels into BufferedImage piece**
- **Add to piece ArrayList**

Each piece is stored at an index in an ArrayList

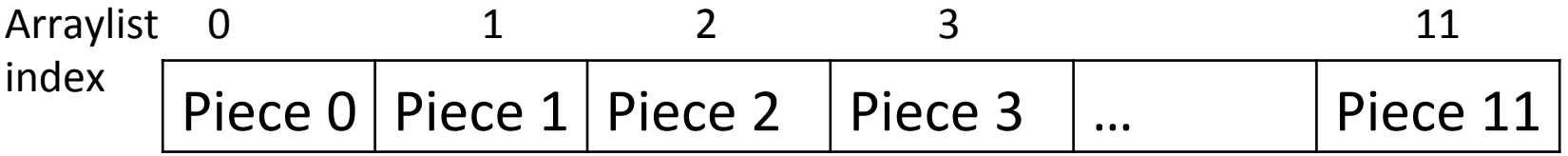
3 x 4 image puzzle pieces

Piece 0	Piece 1	Piece 2	Piece 3
Piece 4	Piece 5	Piece 6	Piece 7
Piece 8	Piece 9	Piece 10	Piece 11

Image is two dimensional

ArrayList

ArrayList is one dimensional



Puzzle.java: shufflePieces() randomly swaps puzzle pieces

```
92  public void swapPieces(int r1, int c1, int r2, int c2) {
93      int index1 = r1*numCols + c1;
94      int index2 = r2*numCols + c2;
95      //swap pieces using temporary variable
96      BufferedImage piece2 = pieces.get(index2); //temporary
97      pieces.set(index2, pieces.get(index1)); //set piece at
98      pieces.set(index1, piece2); //set piece at index1 to b
99  }
100
101 /**
102  * Shuffles the pieces array
103  */
104 private void shufflePieces() {
105     // Simple shuffle: swap each piece with some other one
106     for (int r = 0; r < numRows; r++) {
107         for (int c = 0; c < numCols; c++) {
108             int r2 = (int)(Math.random() * numRows);
109             int c2 = (int)(Math.random() * numCols);
110             swapPieces(r, c, r2, c2);
111         }
112     }
```

Loop over each row and column (nested loop)

Puzzle.java: shufflePieces() randomly swaps puzzle pieces

```
92  public void swapPieces(int r1, int c1, int r2, int c2) {
93      int index1 = r1*numCols + c1;
94      int index2 = r2*numCols + c2;
95      //swap pieces using temporary variable
96      BufferedImage piece2 = pieces.get(index2); //temporary
97      pieces.set(index2, pieces.get(index1)); //set piece at
98      pieces.set(index1, piece2); //set piece at index1 to b
99  }
100
101 /**
102  * Shuffles the pieces array
103  */
104 private void shufflePieces() {
105     // Simple shuffle: swap each piece with some other one
106     for (int r = 0; r < numRows; r++) {
107         for (int c = 0; c < numCols; c++) {
108             int r2 = (int)(Math.random() * numRows);
109             int c2 = (int)(Math.random() * numCols);
110             swapPieces(r, c, r2, c2);
111         }
112     }
```

Loop over each row and column (nested loop)

Randomly choose a piece to swap with piece at row r and column c

Puzzle.java: shufflePieces() randomly swaps puzzle pieces

```
92  public void swapPieces(int r1, int c1, int r2, int c2) {
93      int index1 = r1*numCols + c1;
94      int index2 = r2*numCols + c2;
95      //swap pieces using temporary variable
96      BufferedImage piece2 = pieces.get(index2); //temporary
97      pieces.set(index2, pieces.get(index1)); //set piece at
98      pieces.set(index1, piece2); //set piece at index1 to b
99  }
100
101 /**
102  * Shuffles the pieces array
103  */
104 private void shufflePieces() {
105     // Simple shuffle: swap each piece with some other one
106     for (int r = 0; r < numRows; r++) {
107         for (int c = 0; c < numCols; c++) {
108             int r2 = (int)(Math.random() * numRows);
109             int c2 = (int)(Math.random() * numCols);
110             swapPieces(r, c, r2, c2);
111         }
112     }
```

Loop over each row and column (nested loop)

Randomly choose a piece to swap with piece at row r and column c

Swap piece at (row,col)=(r,c) with piece at (row,col)=(r2,c2)

Puzzle.java: shufflePieces() randomly swaps puzzle pieces

```
92  public void swapPieces(int r1, int c1, int r2, int c2) {
93      int index1 = r1*numCols + c1;
94      int index2 = r2*numCols + c2;
95      //swap pieces using temporary variable
96      BufferedImage piece2 = pieces.get(index2); //temporary
97      pieces.set(index2, pieces.get(index1)); //set piece at
98      pieces.set(index1, piece2); //set piece at index1 to b
99  }
100
101 /**
102  * Shuffles the pieces array
103  */
104 private void shufflePieces() {
105     // Simple shuffle: swap each piece with some other one
106     for (int r = 0; r < numRows; r++) {
107         for (int c = 0; c < numCols; c++) {
108             int r2 = (int)(Math.random() * numRows);
109             int c2 = (int)(Math.random() * numCols);
110             swapPieces(r, c, r2, c2);
111         }
112     }
```

Here we want to swap pieces at different indices in ArrayList

Classic swap pattern

To swap variables a and b:

- **temp = a**
- **a = b**
- **b = temp**

Puzzle.java: shufflePieces() randomly swaps puzzle pieces

```
92  public void swapPieces(int r1, int c1, int r2, int c2) {
93      int index1 = r1*numCols + c1;
94      int index2 = r2*numCols + c2;
95      //swap pieces using temporary variable
96      BufferedImage piece2 = pieces.get(index2); //temporary
97      pieces.set(index2, pieces.get(index1)); //set piece at
98      pieces.set(index1, piece2); //set piece at index1 to b
99  }
100
101 /**
102  * Shuffles the pieces array
103  */
104 private void shufflePieces() {
105     // Simple shuffle: swap each piece with some other one
106     for (int r = 0; r < numRows; r++) {
107         for (int c = 0; c < numCols; c++) {
108             int r2 = (int)(Math.random() * numRows);
109             int c2 = (int)(Math.random() * numCols);
110             swapPieces(r, c, r2, c2);
111         }
112     }
```

Calculate ArrayList index of each piece to swap

Classic swap pattern

To swap variables a and b:

- **temp = a**
- **a = b**
- **b = temp**

Puzzle.java: shufflePieces() randomly swaps puzzle pieces

```
92  public void swapPieces(int r1, int c1, int r2, int c2) {
93      int index1 = r1*numCols + c1;
94      int index2 = r2*numCols + c2;
95      //swap pieces using temporary variable
96      BufferedImage piece2 = pieces.get(index2); //temporary
97      pieces.set(index2, pieces.get(index1)); //set piece at
98      pieces.set(index1, piece2); //set piece at index1 to b
99  }
100
101 /**
102  * Shuffles the pieces array
103  */
104 private void shufflePieces() {
105     // Simple shuffle: swap each piece with some other one
106     for (int r = 0; r < numRows; r++) {
107         for (int c = 0; c < numCols; c++) {
108             int r2 = (int)(Math.random() * numRows);
109             int c2 = (int)(Math.random() * numCols);
110             swapPieces(r, c, r2, c2);
111         }
112     }
```

temp = a

Classic swap pattern

To swap variables a and b:

- **temp = a**
- **a = b**
- **b = temp**

Puzzle.java: shufflePieces() randomly swaps puzzle pieces

```
92  public void swapPieces(int r1, int c1, int r2, int c2) {
93      int index1 = r1*numCols + c1;
94      int index2 = r2*numCols + c2;
95      //swap pieces using temporary variable
96      BufferedImage piece2 = pieces.get(index2); //temporary
97      pieces.set(index2, pieces.get(index1)); //set piece at
98      pieces.set(index1, piece2); //set piece at index1 to b
99  }
100
101 /**
102  * Shuffles the pieces array
103  */
104 private void shufflePieces() {
105     // Simple shuffle: swap each piece with some other one
106     for (int r = 0; r < numRows; r++) {
107         for (int c = 0; c < numCols; c++) {
108             int r2 = (int)(Math.random() * numRows);
109             int c2 = (int)(Math.random() * numCols);
110             swapPieces(r, c, r2, c2);
111         }
112     }
```

a = b

Classic swap pattern

To swap variables a and b:

- **temp = a**
- **a = b**
- **b = temp**

Puzzle.java: shufflePieces() randomly swaps puzzle pieces

```
92  public void swapPieces(int r1, int c1, int r2, int c2) {
93      int index1 = r1*numCols + c1;
94      int index2 = r2*numCols + c2;
95      //swap pieces using temporary variable
96      BufferedImage piece2 = pieces.get(index2); //temporary
97      pieces.set(index2, pieces.get(index1)); //set piece at
98      pieces.set(index1, piece2); //set piece at index1 to b
99  }
100
101 /**
102  * Shuffles the pieces array
103  */
104 private void shufflePieces() {
105     // Simple shuffle: swap each piece with some other one
106     for (int r = 0; r < numRows; r++) {
107         for (int c = 0; c < numCols; c++) {
108             int r2 = (int)(Math.random() * numRows);
109             int c2 = (int)(Math.random() * numCols);
110             swapPieces(r, c, r2, c2);
111         }
112     }
```

b = temp

Classic swap pattern

To swap variables a and b:

- **temp = a**
- **a = b**
- **b = temp**

Review the rest of Puzzle.java and PuzzleGUI.java on your own

Puzzle application

- Puzzle.java does the image processing
- PuzzleGUI.java does the user interaction
- This code is similar to the other code we've been using
- Attend a TA's office hours if these concepts are confusing