

# Images

# Last Time

Attributes/Modes: Function calls that modify all subsequent draw calls.

## Attributes

Change how results look.

- `fill()`
- `stroke()`

## Modes

Change what arguments mean.

- `rectMode()`
- `colorMode()`
- `blendMode()`



# Last Time

Digital color uses an additive model, usually with 8 bits (0-255) each for RGB.

Can use attributes like fill, stroke, and background to affect color of what's drawn.

Multiple ways to use these functions: can specify RGB or pass a color primitive

```
1 color yellow = color(255.0, 255.0, 0.0);  
2 fill(yellow);  
3 rect(0, 0, 200, 200);
```

```
1 fill(255.0, 255.0, 0.0);  
2 rect(0, 0, 200, 200);
```

# Last Time

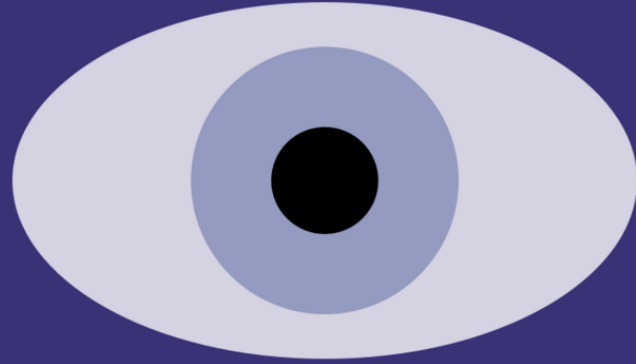
Remember, `draw()` is called in a loop!

```
1 void draw(){  
2   rect(10, 10, 20, 20);  
3   fill(blue);  
4   ellipse(100, 100, 200, 200);  
5 }
```

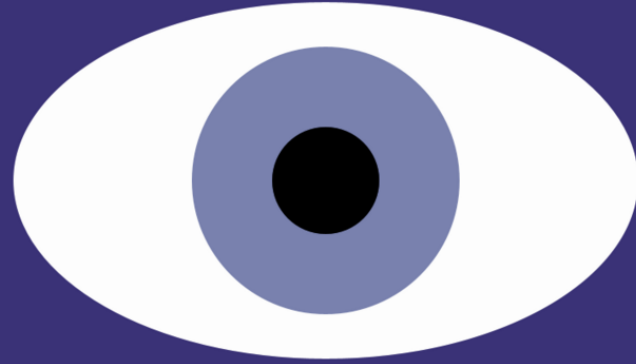
Attributes and modes can bleed through the bottom of the loop and start affecting things before them if you're not careful!

# Code Review

BLEND



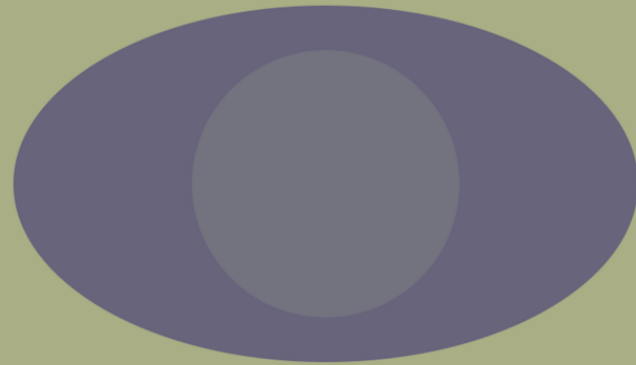
REPLACE



DIFFERENCE



EXCLUSION



# Questions!

# Any tips/tricks for being able to predict where shapes are going to wind up?

Only one I have is to use a `setup()` call which gives you a screen size that has nice divisions. 600 x 600 seems like a good choice.

- 300 = half
- 200 = third
- 150 = quarter
- 100 = sixth

**Do you have ideas?**



# How do places like YouTube manage the size of videos?



What does the next frame look like?



a



b



c



d





# Fundamental idea: the better you can predict something, the less information you need to describe it.



*“ The dog falls in front of the tree.*

*“ The scene suddenly cuts away to several rows of rubber ducks, of various colors, shapes and designs. They appear to be laid out on a long strip of concrete which curves slightly as it moves away from the camera. There are chairs and tables in the background....*



# Can you use hexadecimals for color?

```
1 int doubleInt(int x){
2     return x * 2;
3 }
4
5 void acceptInt(int x){
6     x += 1;
7 }
8
9 void acceptColor(color x){
10    red(x);
11 }
```

```
1 void setup(){
2     int x = 2;
3     acceptInt(2);
4     acceptInt(x);
5     acceptInt(1+1);
6     acceptInt(doubleInt(3));
7
8     color c1 = #bf5700;
9     color c2 = color(255.0, 220.0, 215.0);
10
11    acceptColor(c1);
12    acceptColor(c2);
13    acceptColor(#00ff00);
14    acceptColor(color(2.0, 1.0, 30.0));
15 }
```

Literals, variables, and expressions are all valid instances of a type.

# Lightning Round

**Q: How do you find the color/opacity you're looking for?**

A: Try 'em out and see. I wish there were a better response (though tools like color pickers can make "trying it out" a lot faster). Experience helps with this.

**Q: Can Processing save the image you have when you click "Run"?**

A: Not sure this is what you want, but check out the `save()` function.

# I couldn't set array values outside of `setup()`. Why?

Setting an array value is "doing something." This needs to be done in either `setup()` or `draw()`.

## Why did we need to store colors in an array?

For that hands-on, there was no technical reason to do so. It was more as preparation for what we're going to be doing today.

## < Various questions about scope and globals >

Hang on to these and ask them again after today if it's still unclear.

## **Q: When might it be useful to use explicit RGB instead of a color hex?**

A: If you're generating colors programmatically instead of trying to select colors. For example, if you want to go over all shades of red, it might be better to start off with explicit RGB values.

## **Q: What is up with Crysis?**

A: I enjoy poking fun at it. I find it very funny that CryTek released a game that nobody could actually run and charged \$50 for it.

**Q: Are there any functions that are called multiple times by default like `draw()` is?**

A: Not quite, but hang on to that thought ;)

# Announcement

Project 1 is live!

You can start working on it after today's lecture, but will not have all the components you need to complete it until after Friday's class.

**Due July 18 (Tuesday).**

No sharing code among yourselves, AI models are okay as long as you submit logs.



# Using Color in Processing



Most functions that take color in Processing will have these three variants:

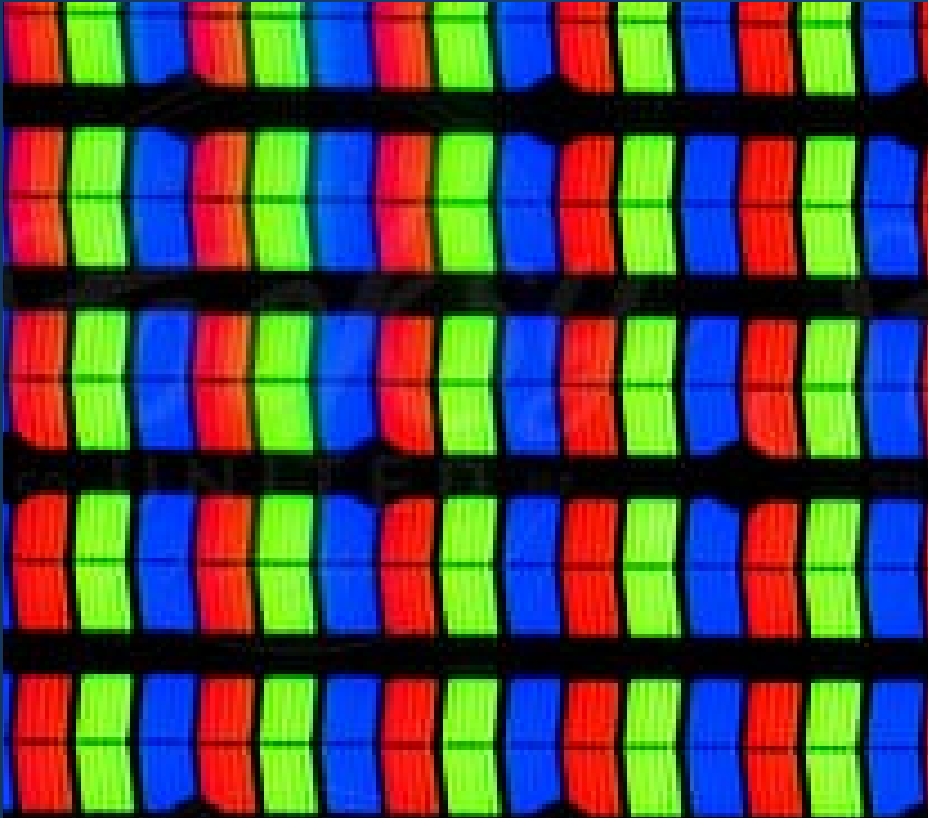
- Take a single color or hex argument
- Take a single numerical grayscale argument
- Take three separate arguments (RGB)

and for each of these, there will usually be a variant that takes an additional alpha (transparency) argument.

# More About Colors

# RGB Color

We have defined color as a triplet of numbers, representing the red, green, and blue components of the color.



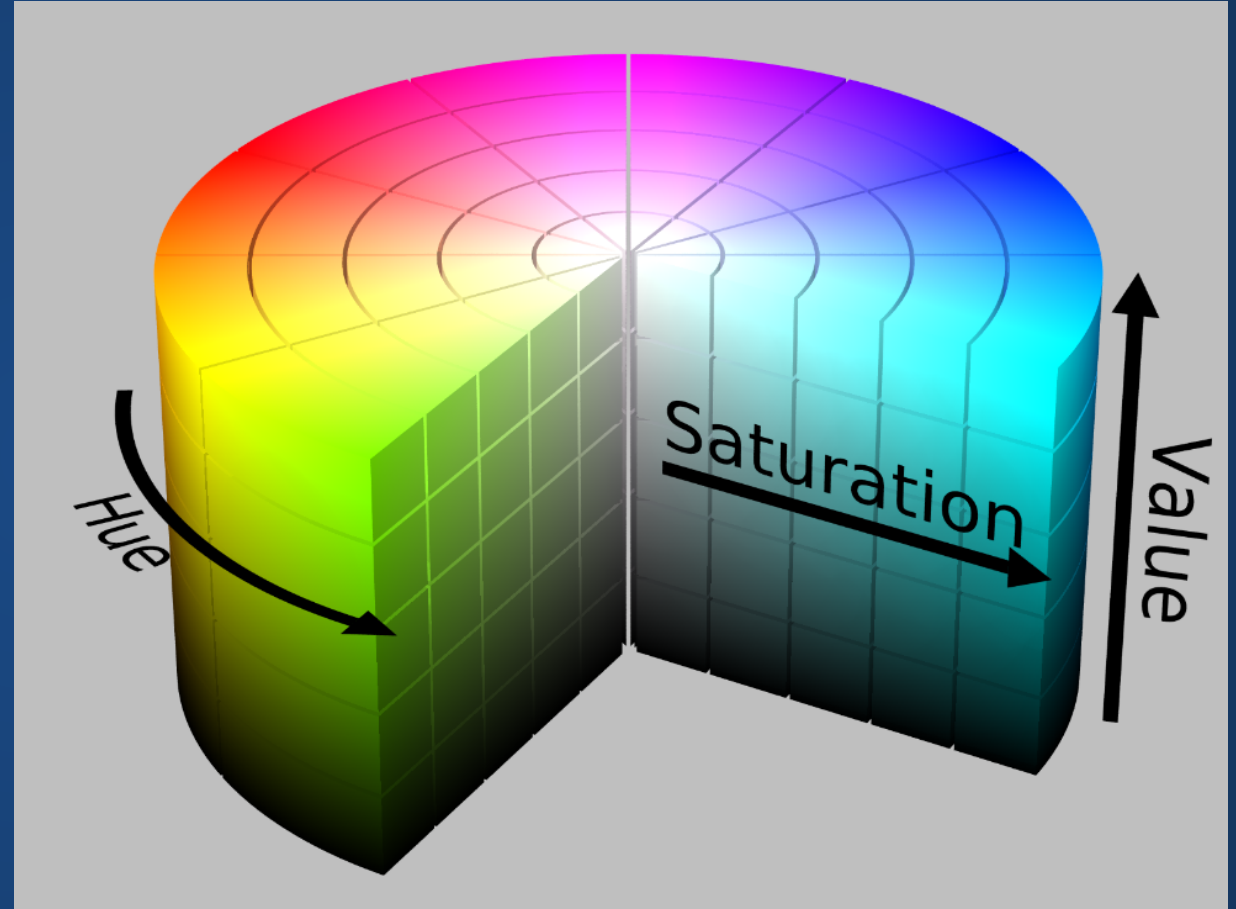
```
1 color yellow = color(255.0, 255.0, 0.0);
```

What if I told you there was another way?

# HSV/HSB

Hue-Saturation-Value (or Brightness) is commonly used in color pickers

- Hue: pure color
- Saturation: amount of color
- Value: darkness or lightness of color



# How do we change the interpretation of colors between HSV/RGB?

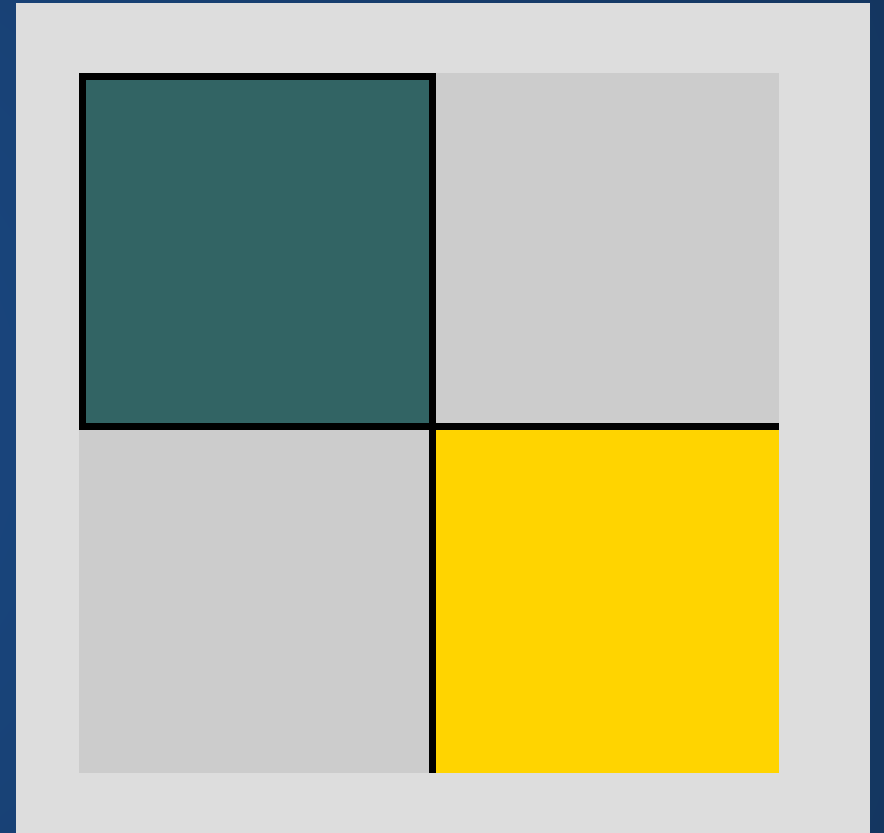
```
1 colorMode(RGB, 255, 255, 255);  
2 colorMode(HSB, 360, 100, 100);  
3 colorMode(RGB, 1.0, 1.0, 1.0);  
4 colorMode(HSB, 100);
```

# Extracting Data from a color

```
1 float r = red(color c);  
2 float g = green(color c);  
3 float b = blue(color c);
```

```
1 float h = hue(color c);  
2 float s = saturation(color c);  
3 float v = brightness(color c);
```

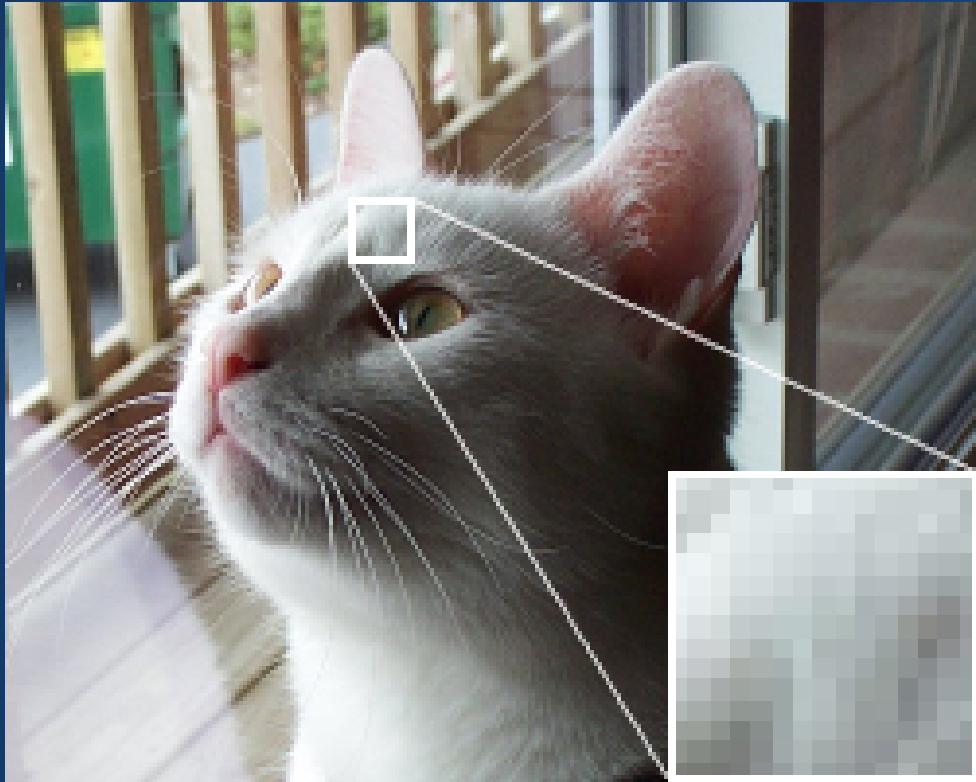
```
1 colorMode(RGB, 255, 255, 255);
2 fill(50, 100, 100);
3 rect(0, 0, 50, 50); //Rect1
4 colorMode(HSB, 360, 100, 100);
5 fill(50, 100, 100);
6 rect(50, 50, 50, 50); //Rect2
```



# Image Storage



# Pixels



Each pixel has some bits that determine the color.



Images are composed of pixels.

# Image Buffer

Screen pixel data is stored in a buffer (array), which allows us to access per-pixel information.

## How the pixels look on screen

0	1	2	3	4	5
6	7	8	9	10	11
12	13	14	15	16	17
18	19	20	21	22	23
24	25	26	27	28	29
30	31	32	33	34	35

## How the pixels are stored in memory



# Images in Processing

- Images in processing are stored in the `PImage` type.
- `PImage` allows us to load and display data.
- We can also manipulate data:
  - Size
  - Position
  - Opacity
  - Tint
- `image(PImage img, float x, float y, float width, float height)` to display an image.
- Use `loadImage(string fname)` to load an image.

# Displaying an Image

```
1 PImage img;
2
3 void setup(){
4     img = loadImage("foo.png");
5     size(100, 100);
6 }
7
8 void draw(){
9     image(img, 0, 0);
10 }
```

# Adjusting Window to Image Size

```
1 void setup(){
2     surface.setResizable(true);
3     img = loadImage("foo.png");
4     surface.setSize(img.width, img.height);
5 }
```

# Modifying Pixels

In principle, we can modify pixel data just by modifying the `pixels` member of `PImage`.

```
1 PImage img;
2
3 void setup(){
4     // Do setup + loading
5 }
6
7 void draw(){
8     for (int i = 0; i < img.pixels.length; i++){
9         color c = color(/* Some initialization */);
10        img.pixels[i] = c;
11    }
12 }
```

**But this won't always work!**

# Modifying Pixels

Processing maintains **two** copies of the data in the image:

- One is stored in the PImage variable
- The other is used to actually display the image.



PImage Data



Screen Data



PImage Data

loadPixels()



updatePixels()



Screen Data

```
1 PImage img;
2
3 void setup(){
4   // Do setup + loading
5 }
6
7 void draw(){
8   img.loadPixels(); // Read updated pixel data into img
9   for (int i = 0; i < img.pixels.length; i++){
10    color c = color(/* Some initialization */);
11    img.pixels[i] = c;
12  }
13  img.updatePixels(); // Write updated img data to screen
14 }
```

Image manipulation *might* not work without these calls (depends on OS)



# How can we access a pixel by its (x,y) value?

How the pixels look on screen

0	1	2	3	4	5
6	7	8	9	10	11
12	13	14	15	16	17
18	19	20	21	22	23
24	25	26	27	28	29
30	31	32	33	34	35

How the pixels are stored in memory

0	1	2	3	4	5	6	7	8	9
---	---	---	---	---	---	---	---	---	---

- Perform a *stride* into the 1D array to find the row we're looking for.
- Then use the column to find the final placement in the 1D array.
- We need to know the image width to do this.

0	1	2	3	4	5
---	---	---	---	---	---

0
1
2
3
4
5

0	1	2	3	4	5
6	7	8	9	10	11
12	13	14	15	16	17
18	19	20	21	22	23
24	25	26	27	28	29
30	31	32	33	34	35

Where is (3,5)?

Where is (5,2)?

Where is (2,0)?

**Where is (x,y)?**

# Traversing by (x,y)

```
1 img.loadPixels();
2 for(int x = 0; x < img.width; x++){
3     for(int y = 0; y < img.height; y++){
4         int index = x + y * img.width;
5         color c = img.pixels[index];
6     }
7 }
```

# Tint

`tint()` modifies the color of a displayed image.  
`noTint()` disables tint modifications.

```
1 size(400,400);  
2 PImage img;  
3 img = loadImage("yuya-onsen.jpg");  
4 image(img, 0, 0);  
5 tint(0, 153, 204); // Tint blue  
6 image(img, width/2, 0);
```



# The Data Directory

Processing needs one of two things to be true before it loads an image:

1. You need to give an *absolute* path to the file, i.e. a filepath that starts with '/' on MacOS + Linux, or a drive letter on Windows.
2. The file needs to be in a data directory within the Processing sketch.

- Processing Project Directory

- your\_program.pde
- other\_code.pde
- data
  - your\_image.jpeg
  - your\_data.json

```
1 PImage img1;
2
3 void setup(){
4     img1 = loadImage("your_image.jpeg");
5 }
```

# Hands-On: Creating Tint

Recreate Processing's `tint()` function using a method you create called `myTint()`. Do not use the existing `tint()` method.

1. The method should take RGB data. (OPTIONAL): make it so that you can take either separate RGB arguments or a single color with the same function.
2. `myTint()` should be "per image" rather than "per-screen" so you should have an argument for the `PImage` you will tint.
3. You do not need to worry about undoing the tint.

If you don't have a suitable picture, check the `#animals-are-cute` channel on EdStem.

Take the directory which contains your `.pde` file and your data directory, and ZIP that into an archive. Submit this archive to Canvas.

# Local Transformations



# How do you give a function default/optional arguments?

```
1 def say_a_thing(thing="Hello!"):  
2     print(f"The computer says {thing}")
```

```
1 say_a_thing()
```

The diagram consists of three white rectangular boxes on a dark blue background. At the top center is a box containing the function definition for `say_a_thing`. Below it, on the left, is a box with the function call `say_a_thing()`. On the right is a box with the function call `say_a_thing("Moo.")`. Two white arrows originate from the bottom corners of these two lower boxes and point upwards towards the bottom corners of the top box, indicating that these calls are instances of the function defined above.

```
1 say_a_thing("Moo.")
```



# Overloads

Must differ in number or type of arguments.

```
1 void sayAThing(){
2     println("The computer says Hello!");
3 }
4
5 void sayAThing(String thing){
6     println("The computer says " + thing);
7 }
```

```
1 sayAThing();
```



```
1 sayAThing("Moo.");
```



```
1 // Can use this instead of the first overload of
2 // sayAThing---but we cannot have both at once,
3 // since those would have the same number + type
4 // of arguments
5
6 void sayAThing(){
7     sayAThing("Hello!");
8 }
```

# Overloads can be used for more than default arguments!

```
1 String smushTogether(String a, String b){
2     return a + b;
3 }
4
5 PImage smushTogether(PImage a, PImage b){
6     PImage result = createImage(a.width, 2 * a.height, ARGB);
7     int imageSize = a.pixels.length;
8     for(int i = 0; i < imageSize; i++){
9         result.pixels[i] = a.pixels[i];
10    }
11    for(int i = 0; i < imageSize; i++){
12        result.pixels[i + imageSize] = b.pixels[i];
13    }
14    return result;
15 }
16
17 int smushTogether(int a, int b){
18     String s_a = String.valueOf(a);
19     String s_b = String.valueOf(b);
20     String squished = smushTogether(s_a, s_b);
21     return parseInt(squished);
22 }
```

You can look up "Java method overloading" if you want to learn more.

# Why couldn't we tint our image by laying a transparent rect over top?

```
1 void setup(){
2   img = loadImage("oogabooga.png");
3   myTint(img, 100, 200, 75);
4 }
5
6 void draw(){
7   image(img, 50, 30);
8 }
```

myTint() has to draw the rectangle here, but how do we know where to draw it?

What if we move the image while drawing it?



# Code Review

# The Values In Your Neighborhood

They're the values that you meet when you're walking down the street...

The manipulations we have seen so far are per-pixel. Output pixel values only depend on the input pixel values.

Example: Grayscale image. How can we find a single value that captures the information of three color channels?





Some manipulations are **local**: require information about neighboring pixels as well.

Example: Increase contrast between pixels.





# Image Kernel

Can go by several names:

- Kernel
- Convolution Matrix
- Mask

**A small matrix which is used to make adjustments to image data based on local information.**

Almost always small (3x3, 5x5) and square.

# Convolution

1. Multiply corresponding cells
2. Sum the resulting values

NOT a matrix multiply!!

39	33	35	36	31
35	34	36	33	34
34	33	<b>36</b>	34	32
32	36	35	36	35
33	31	34	31	32

$$\otimes \begin{array}{|c|c|c|} \hline 1 & 2 & 1 \\ \hline 2 & 4 & 2 \\ \hline 1 & 2 & 1 \\ \hline \end{array} = \left\{ \begin{array}{ccc} 1 \cdot 34 & 2 \cdot 36 & 1 \cdot 33 \\ 2 \cdot 33 & 4 \cdot 36 & 2 \cdot 34 \\ 1 \cdot 36 & 2 \cdot 35 & 1 \cdot 36 \end{array} \right\} = \{139 + 278 + 142\} = 559$$

# Reasoning About Kernels

# What Does This Kernel Do?

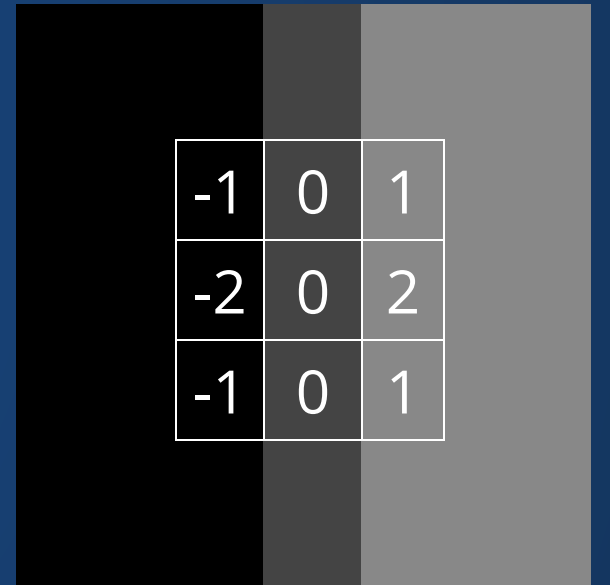
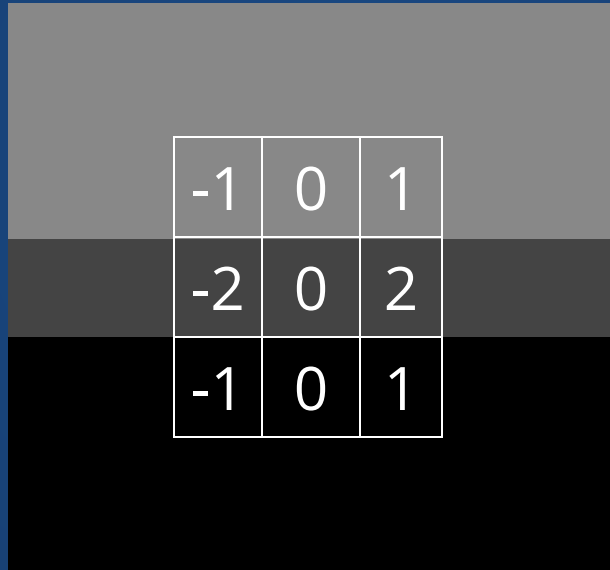
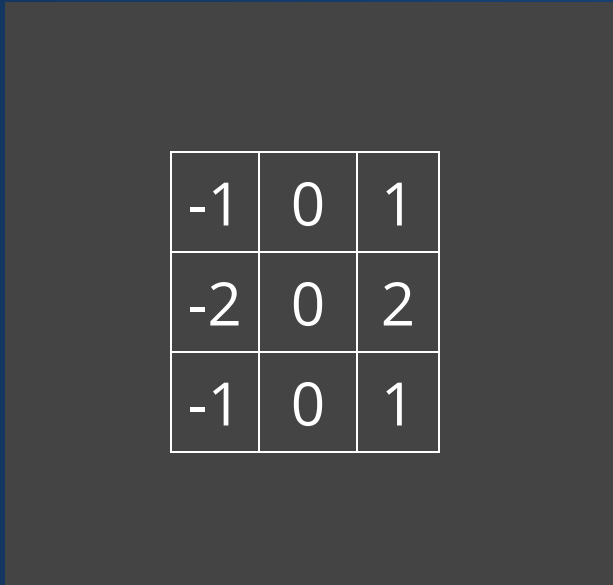
$$\frac{1}{9} \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$$



39	33	35	36	31
35	34	36	33	34
34	33	<b>36</b>	34	32
32	36	35	36	35
33	31	34	31	32

$$\otimes \begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix} = \left\{ \begin{array}{ccc} 1 \cdot 34 & 2 \cdot 36 & 1 \cdot 33 \\ 2 \cdot 33 & 4 \cdot 36 & 2 \cdot 34 \\ 1 \cdot 36 & 2 \cdot 35 & 1 \cdot 36 \end{array} \right\} = \{139 + 278 + 142\} = 559$$

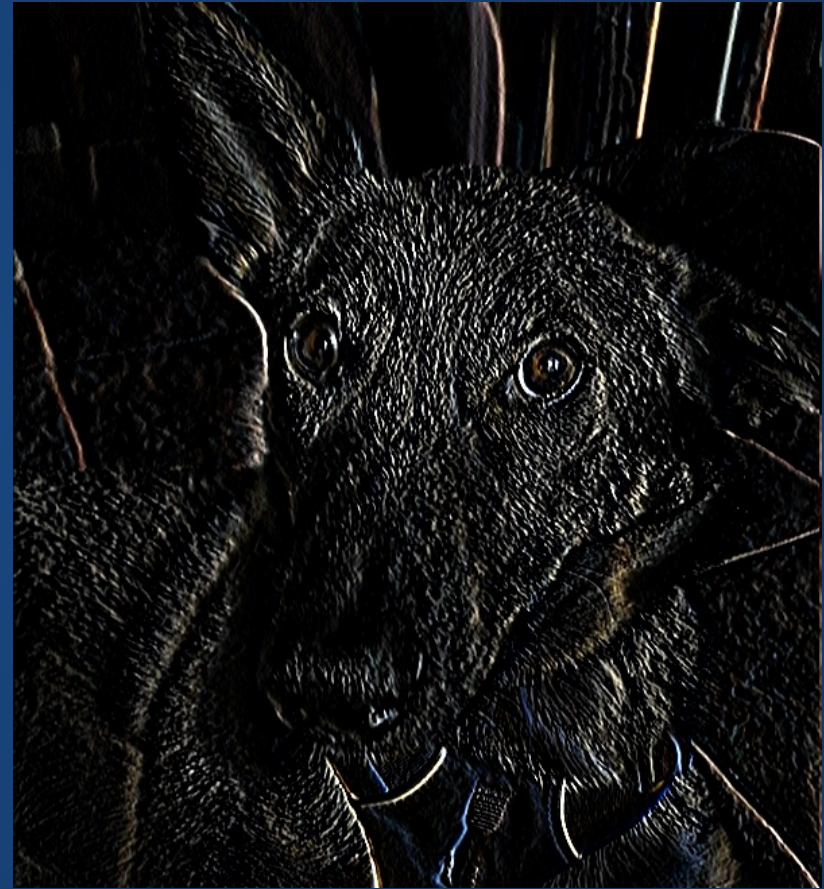
-1	0	1
-2	0	2
-1	0	1



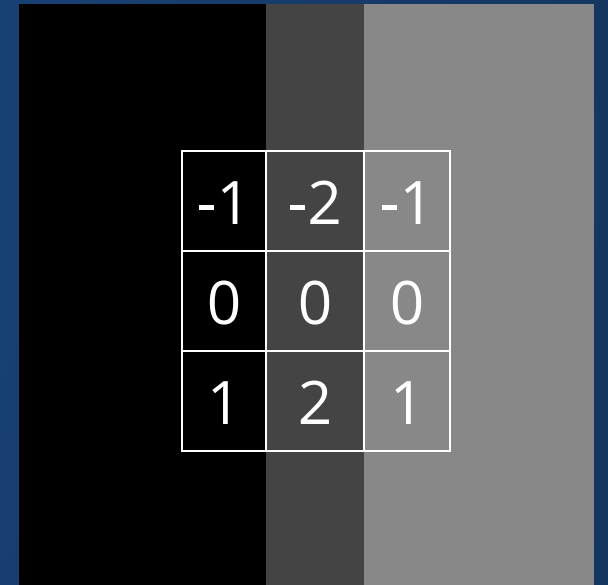
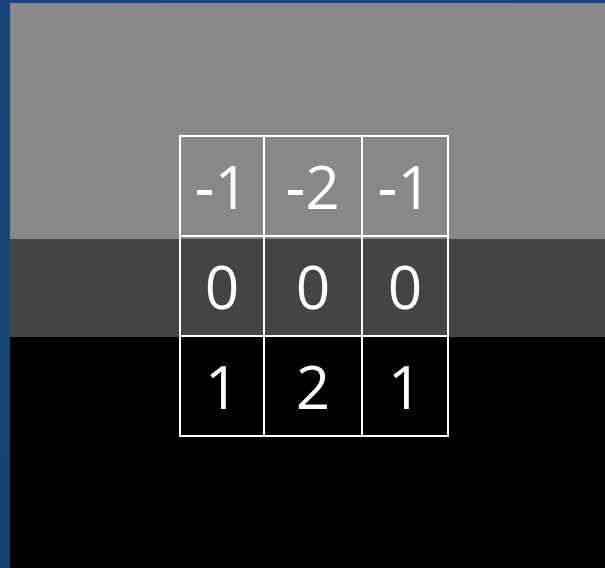


Blacks out the result unless there is a vertical edge in the image. This is known as a *Sobel filter* or an edge-detection filter.

-1	0	1
-2	0	2
-1	0	1



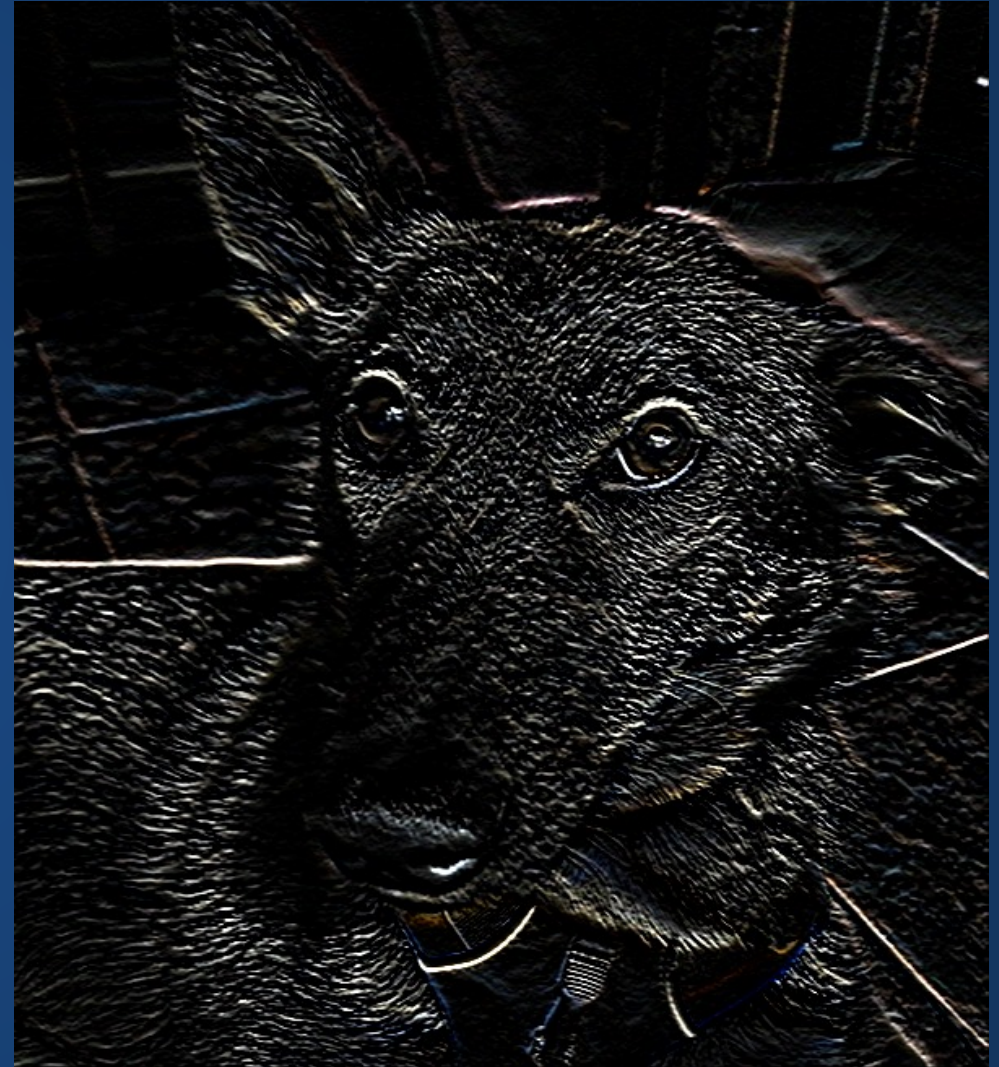
-1	-2	-1
0	0	0
1	2	1





# Horizontal edge detector

-1	-2	-1
0	0	0
1	2	1





-1	0	1
-2	0	2
-1	0	1



-1	-2	-1
0	0	0
1	2	1





$$\frac{1}{16}$$

1	2	1
2	4	2
1	2	1

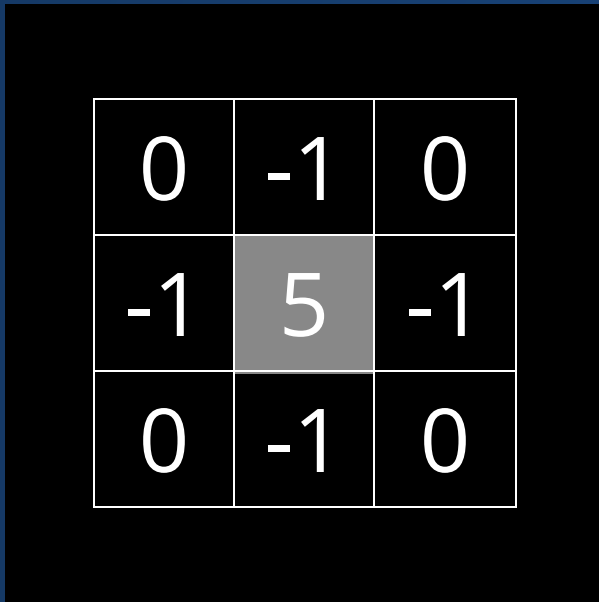


Known as a Gaussian blur

# What does this kernel do?

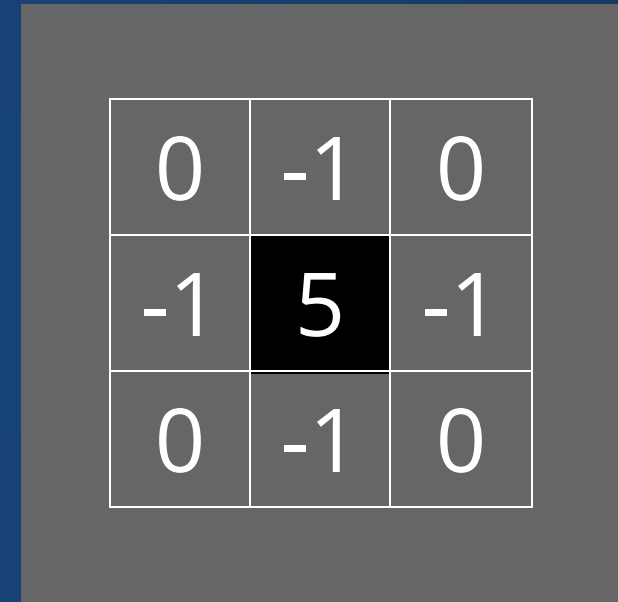
0	-1	0
-1	5	-1
0	-1	0

If center pixel is brighter than others, boost its brightness.



0	-1	0
-1	5	-1
0	-1	0

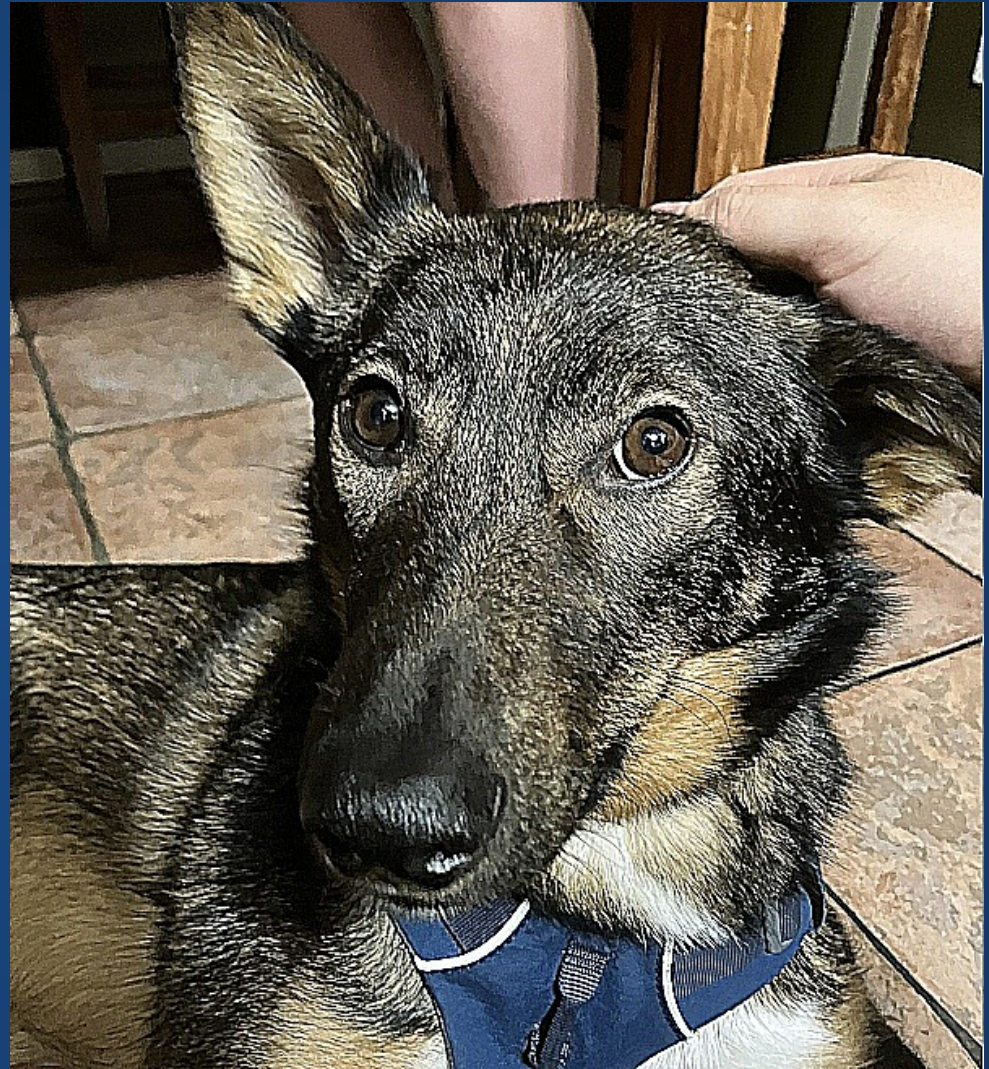
If center pixel is darker than others, lower its brightness.



0	-1	0
-1	5	-1
0	-1	0

**Effect: emphasizes pixels that are brighter than their neighbors.**







# Applying Convolutions

39	33	35	36	31
35	34	36	33	34
34	33	<b>36</b>	34	32
32	36	35	36	35
33	31	34	31	32

$$\otimes \begin{array}{|c|c|c|} \hline 1 & 2 & 1 \\ \hline 2 & 4 & 2 \\ \hline 1 & 2 & 1 \\ \hline \end{array} = \left\{ \begin{array}{ccc} 1 \cdot 34 & 2 \cdot 36 & 1 \cdot 33 \\ 2 \cdot 33 & 4 \cdot 36 & 2 \cdot 34 \\ 1 \cdot 36 & 2 \cdot 35 & 1 \cdot 36 \end{array} \right\} = \{139 + 278 + 142\} = 559$$



# Step 1: Apply kernel to single pixel

```
1 void applyKernelTo(PImage img, int x, int y){  
2     // ?  
3 }
```

39	33	35	36	31
35	34	36	33	34
34	33	<b>36</b>	34	32
32	36	35	36	35
33	31	34	31	32

⊗

1	2	1
2	4	2
1	2	1

Need to access neighborhood of x, y to obtain image values, and same for kernel.

```

1 float[] kernel = {0, -1, 0, -1, 5, -1, 0, -1, 0};
2
3 void applyKernelTo(PImage img, int x, int y){
4     for(int x_off = -1; x_off <= 1; x_off++){
5         for(int y_off = -1; y_off <= 1; y_off++){
6
7             }
8     }
9 }

```

39	33	35	36	31
35	34	36	33	34
34	33	<b>36</b>	34	32
32	36	35	36	35
33	31	34	31	32

⊗

1	2	1
2	4	2
1	2	1

39	33	35	36	31
35	34	36	33	34
34	33	<b>36</b>	34	32
32	36	35	36	35
33	31	34	31	32

1	2	1
2	4	2
1	2	1

```

1 float[] kernel = {0, -1, 0, -1, 5, -1, 0, -1, 0};
2
3 void applyKernelTo(PImage img, int x, int y){
4     for(int x_off = -1; x_off <= 1; x_off++){
5         for(int y_off = -1; y_off <= 1; y_off++ ){
6             int img_index = (y + y_off) * img.width + (x + x_off);
7             int ker_index = (1 + y_off) * 3 + (1 + x_off);
8
9             // ?
10        }
11    }
12 }

```

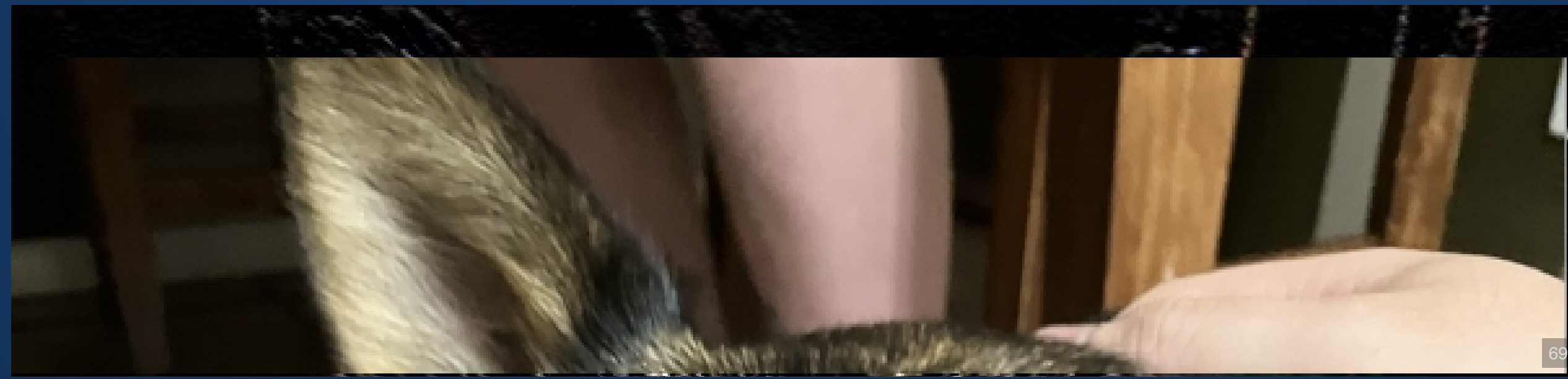
```

1 float[] kernel = {0, -1, 0, -1, 5, -1, 0, -1, 0};
2
3 void applyKernelTo(PImage img, int x, int y){
4     // Declare and initialize final_red, final_green, final_blue
5     for(int x_off = -1; x_off <= 1; x_off++){
6         for(int y_off = -1; y_off <= 1; y_off++){
7             int img_index = (y + y_off) * img.width + (x + x_off);
8             int ker_index = (1 + y_off) * 3 + (1 + x_off);
9
10            float red = img.pixels[img_index];
11            final_red += red * kernel[ker_index];
12
13            // Perform similar operations for green and blue;
14        }
15    }
16    final_red = constrain(red, 0, 255); // Why?
17    // Also clamp green and blue
18    color final_color = color(final_red, final_green, final_blue);
19    img[x + img.width * y] = final_color;
20 }

```

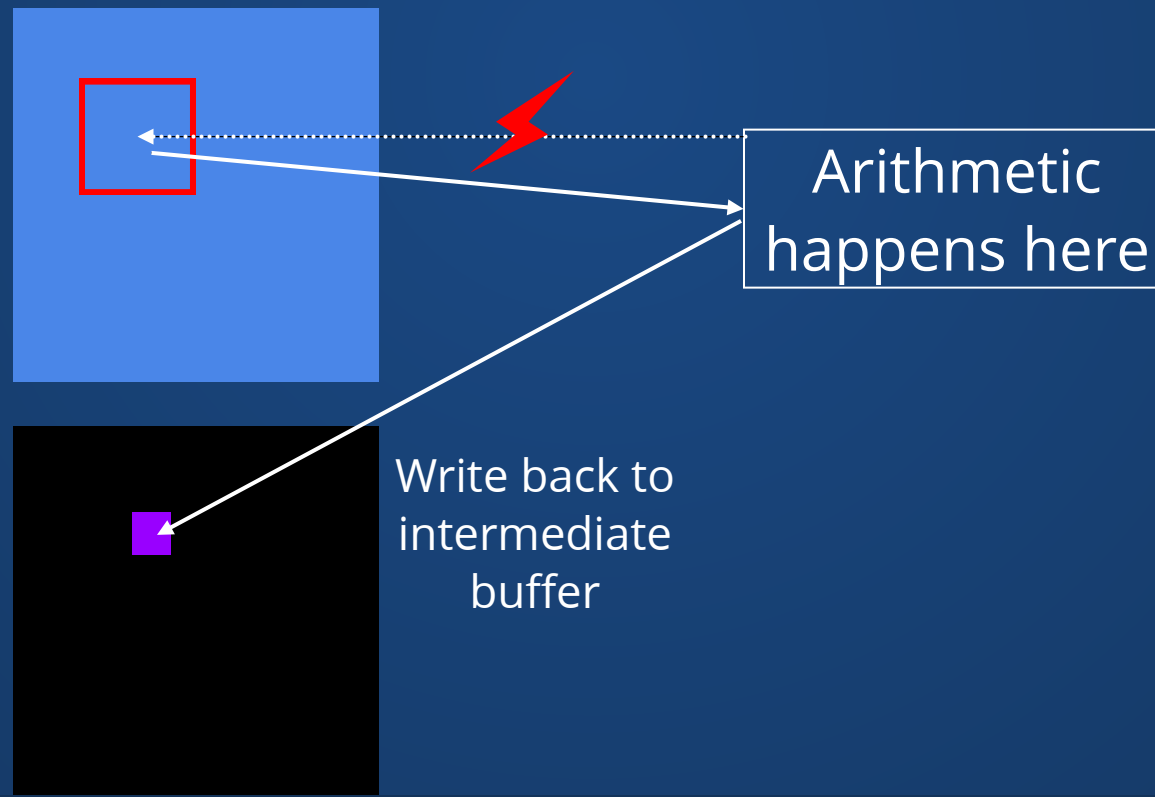


# Why can't we store the new value in the existing image?



# Intermediate Buffer

- Array of pixels which matches the size of the image
- Provides a safe location for storing image data
- Allows program to preserve original image data
- Common trick to increase speed of rendering (double buffering)





# Creating a Buffer

There are a few ways to create a duplicate image.

- The simplest (when possible) is just to load the image twice:

```
PImage img = loadImage(image_file);  
PImage buf = loadImage(image_file);
```

- You can also create a blank image:

```
PImage buf = createImage(width, height, ARGB);
```

- If necessary, copy pixel values between the images:

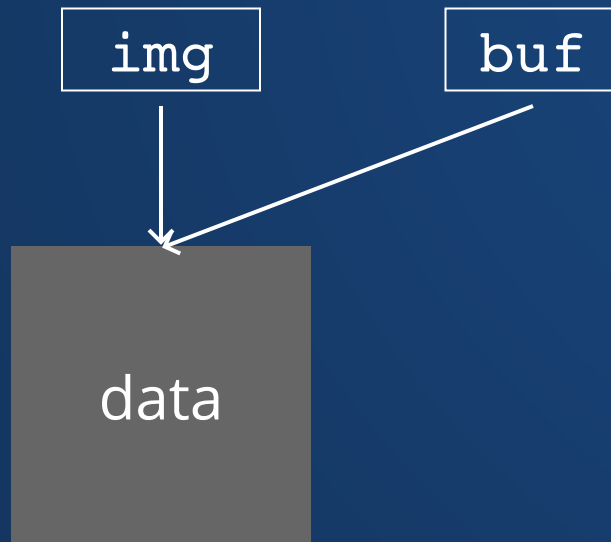
```
copy(img, x, y, width, height, x, y, width,  
height);
```

# Beginner's Trap

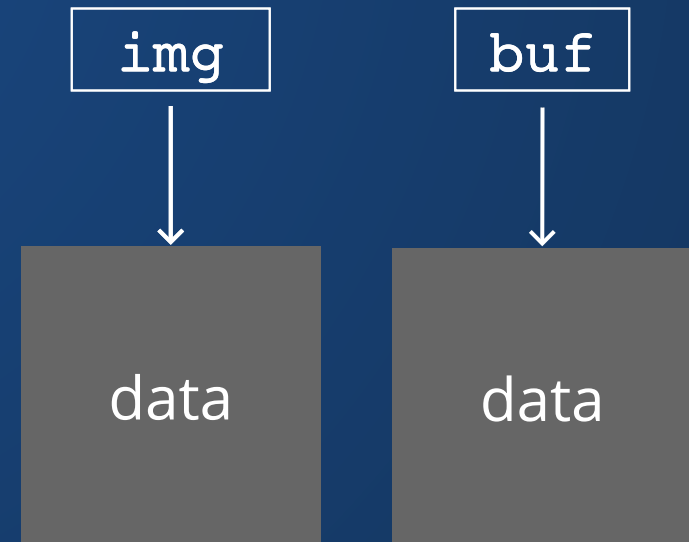
```
PImage img = loadImage(image_file);  
PImage buf = img;
```

This does **not** create a full copy of the image! This is called a *shallow copy* of the data. Use the techniques on the previous slide to get a deep copy.

## Shallow Copy



## Deep Copy

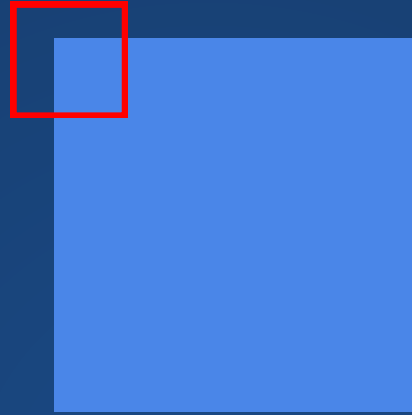


# So Far

```
1 float[] kernel = {0, -1, 0, -1, 5, -1, 0, -1, 0};
2
3 void applyKernelTo(PImage img, int x, int y){
4     // Declare and initialize final_red, final_green, final_blue
5     for(int x_off = -1; x_off <= 1; x_off++){
6         for(int y_off = -1; y_off <= 1; y_off++){
7             int img_index = (y + y_off) * img.width + (x + x_off);
8             int ker_index = (1 + y_off) * 3 + (1 + x_off);
9
10            float red = img.pixels[img_index];
11            final_red += red * kernel[ker_index];
12
13            // Perform similar operations for green and blue;
14        }
15    }
16    final_red = constrain(red, 0, 255); // Why?
17    // Also clamp green and blue
18    color final_color = color(final_red, final_green, final_blue);
19    img[x + img.width * y] = final_color; // Broken
20 }
```

- Use function to compute the convolution at a single pixel location.
- Within function, use for-loops over offsets to compute result of convolution.
- Use 2D array math to compute indices into image and kernel.
- Read data from image and compute convolution value.
- Clamp final values into valid range.
- Write final value to intermediate buffer (you will need to change the code to accomplish this!).
- Call this function on every location in the image.

# Literal Edge Cases



```
applyKernelTo(img, 0, 0)
```

Strategies for dealing with this:

- Don't apply convolution to edges or corners
- Missing values are filled in with a default (0 or 255)
- Wrap missing pixels from the other side of the image
- Mirror missing pixels from the other side of the kernel

# Hands-On: Image Kernels

1. Take your "sharpen" kernel and store it in a variable.
2. Create a new image buffer to store the final, convolved image data.
3. Apply the sharpen kernel to an image and store the convolved data into your secondary image buffer.
4. Display the convolved buffer to the screen.



# Index Cards!

1. Your name and EID.
2. One thing that you learned from class today. You are allowed to say "nothing" if you didn't learn anything.
3. One question you have about something covered in class today. You *may not* respond "nothing".
4. (Optional) Any other comments/questions/thoughts about today's class.