# Objective

You will work with bitmap (aka *raster*) graphics, modifying the pixels to produce effects like tinting, negating, or averaging color values. You will also perform operations like moving or swapping pixels.

In this lab, you will write several short classes that implement an *interface*. An interface allows the programmer to specify common behaviors among objects that are not related through inheritance. In this project, we will use two interfaces, which are provided in the code:

interface PixelFilter

{

color **convert(**color inColor**)**;

}

interface ImageFilter

{

void **convert(**PImage img**)**;

}

## Topics: color, arrays, bitmap graphics, algorithms, interfaces

# Instructions

In the provided sketch, you will find a fair bit of already written code. In the main tab (***lab\_17\_image\_filters***) you will see code that loads in an image, and creates two PImage objects from that bitmap graphic. There is also a method that allows you to select different filters that can be applied to the image.

We will write classes that implement an ImageFilter interface. This filter is intended to modify the pixels in a PImage object in some way. For example, we will create a filter that flips an image horizontally or vertically.

We will also write classes that modify the color value of individual pixels within an image. They will implement the PixelFilter interface. Object that implement this interface are “plugged in” to an ImageFilter to create an easy way to modify images based on mathematically adjusting color values.

In the ***Example\_Filters*** tab you will find a complete class called BasicImageFilter. When a BasicImageFilter is instantiated, it requires a PixelFilter object to be supplied. The BasicImageFilter uses the supplied PixelFilter to process each pixel in the image.

## Exercise 1 (Part\_01)

The brightness of a pixel can be determined by averaging the values of the three color components that make up the pixel (the red, green, and blue values).

**brightness = (red + green + blue) / 3**

The calculation will yield a value between 0 (darkest) and 255(brightest).

A grayscale color can be created by making a color whose red, green, and blue components are equal. For example, **color(**0, 0, 0**)** creates black, and **color(**255, 255, 255**)** creates white. Values in-between the extremes create different shades of gray. Because this is such a common operation, Processing provides an alternate version of the **color()** method that takes a single value. That value is used for all three color channels. For example, white is **color(**255**)** and black is **color(**0**)**.

If we measure the brightness of each pixel in an image, then create a new color using the same brightness value for all three components, we create a grayscale version of the image.

In the **convert()** method of the GrayscaleFilter, you will add code to return the color passed in via the inColor parameter as an equivalent grayscale color. You will need to make use of the **red(**color**)**, **green(**color**)** and **blue(**color**)** methods available in Processing to pull the channel values out of the parameter variable. Those three methods return a float representing the brightness of the respective channel in a given color.

## Exercise 2 (Part\_01)

A negative color is created by subtracting the individual color components (red, green, and blue) from 255 (the brightest intensity) and creating a new color from the results of the calculation:

originalRed = **red(**originalColor**)**; //Assume red is 200, for example

newRed = 255 – originalRed; //So newRed is 55 in this case

//Do the same for green and blue

color newColor = **color(**newRed, newGreen, newBlue**)**;

This makes sense if you think it through. The negative of black (brightness value 0) is white (value 255). The formula is 255 – 0 = 255, so creating the negative of black results in white.

The negative of white is black, and 255 – 255 = 0, so it works in that direction, too.

Now, add the required code to the **NegativeFilter** class, and test it to see if you made it work correctly.

## Exercise 3 (Part\_01)

Tinting an image is a simple process. To tint an image, you create a new color by multiplying the color channels by a scale factor. Each component (red, green, blue) gets a multiplier. Through a clever choice of scale factors, you can remove all the components of a color except one. For example, to see only the green part of an image, make aTintFilterthat multiplies the red and blue channel by 0.0, and the green channel by 1.0. You could also brighten an image by using equal scale factors for each channel, all greater than 1.0.

The TintFilter class will differ from the GrayscaleFilter and NegativeFilter in one important aspect: it needs some instance data. The TintFilter will need to know the scale values for each Color component. This will be passed in to the constructor, and used in the **convert()** method.

TintFilter brighter = new **TintFilter**(1.5, 2.0, 1.7**)**;

color newColor = brighter.**convert( color**(200, 38, 126**)** **)**;

In this example, newColor would have the values (255, 76, 214), a brighter color than the original.

Edit the TintFilter class, and fill in the missing code. Test it to be sure it works.

## Exercise 4 (Part\_02)

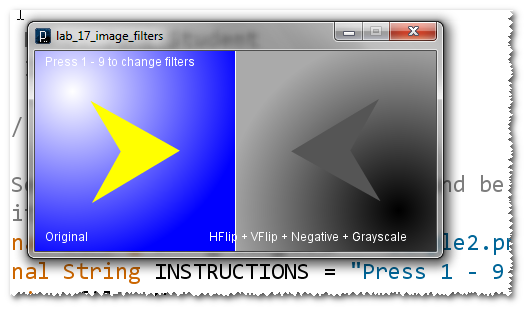
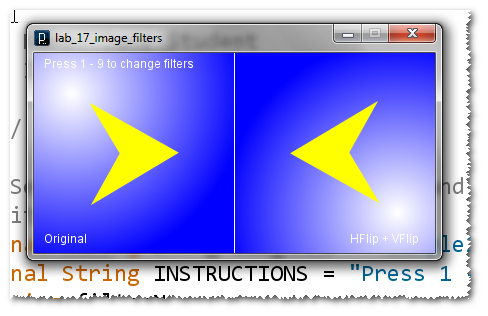
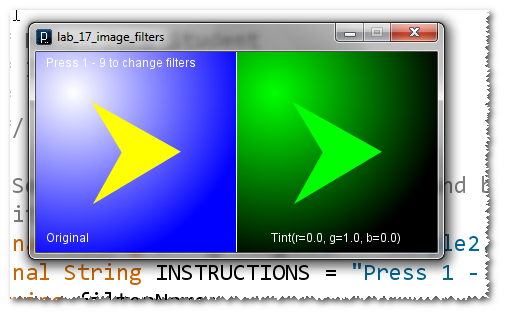
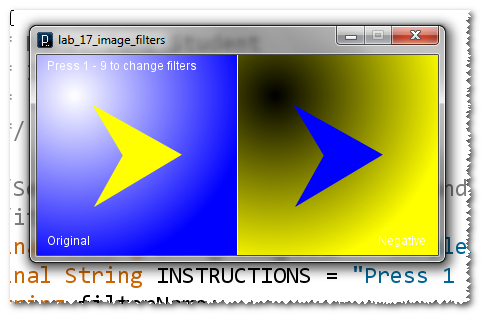
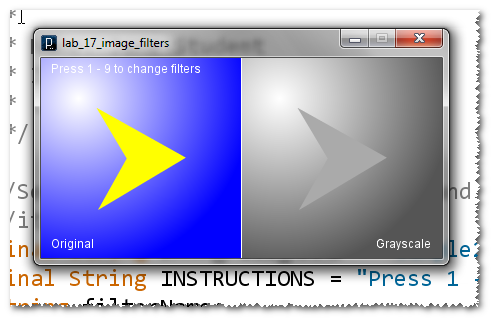
The next exercise is to create two ImageFilters. One will be called HFlipFilter, and it will flip the image horizontally (i.e. rotate around the y-axis). The other filter will be called VFlipFilter, and it will flip the image vertically (i.e. rotate around the x-axis). You will have to come up with the method for flipping the image.

## Exercise 5 (Part\_03)

The final task is to create a PixelFilter and ImageFilter that stores references to any two other filters. When the ComboPixelFilter or ComboImageFilter is used, it processes the input through both of its stored filters.

Interestingly, a combining filter could combine two other combining filters, effectively creating a quad-filter! This can allow you to easily create compound effects using existing filter classes as already written, no changes necessary.

# Examples



# Challenge

Create your own filters! There are many possibilities:

* make a vignette effect, which is a slight darkening at the corners of an image;
* make a green-screen effect by removing all pixels close to a certain color value, then showing the pixels of a different image in those places;
* create a lens-flare centered around a certain point in the image;
* create a mirror effect by slicing the image in half, then flipping one half over the new axis;
* create a pointillism effect by “magnifying” some pixels into larger circles, and omitting others