

Processing:
a programming
handbook for
visual designers
and artists

Casey Reas
Ben Fry

The MIT Press
Cambridge, Massachusetts
London, England

Color 1: Color by Numbers

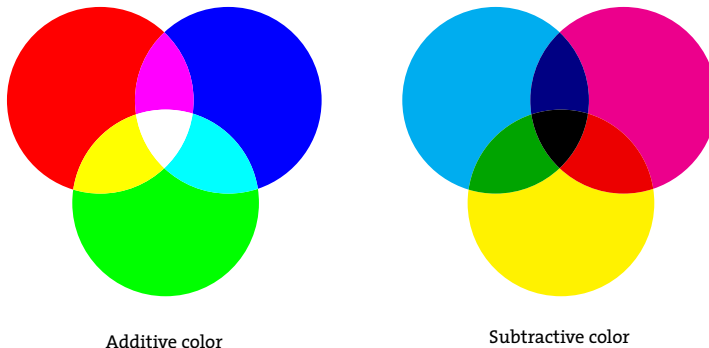
This unit introduces code elements and concepts for working with color in software.

Syntax introduced:

`color`, `color()`, `colorMode()`

When Casey and Ben studied color in school, they spent hours carefully mixing paints and applying it to sheets of paper. They cut paper into perfect squares and carefully arranged them into precise gradations from blue to orange, white to yellow, and many other combinations. Over time, they developed an intuition that allowed them to achieve a specific color value by mixing the appropriate components. Through focused labor, they learned how to isolate properties of color, understand the interactions between colors, and discuss qualities of color.

Working with color on screen is different from working with color on paper or canvas. While the same rigor applies, knowledge of pigments for painting (cadmium red, Prussian blue, burnt umber) and from printing (cyan, yellow, magenta) does not translate into the information needed to create colors for digital displays. For example, adding all colors together on a computer monitor produces white, while adding all the colors together with paint produces black (or a strange brown). A computer monitor mixes colors with light. The screen is a black surface, and colored light is added. This is known as additive color, in contrast to the subtractive color model for inks on paper and canvas. This image presents the difference between these models:



The most common way to specify color on the computer is with RGB values. An RGB value sets the amount of red, green, and blue light in a single pixel of the screen. If you look closely at a computer monitor or television screen, you will see that each pixel is comprised of three separate light elements of the colors red, green, and blue; but because our eyes can see only a limited amount of detail, the three colors mix to create a single color. The intensities of each color element are usually specified with values between 0 and 255 where 0 is the minimum and 255 is the maximum. Many software applications

also use this range. Setting the red, green, and blue components to 0 creates black. Setting these components to 255 creates white. Setting red to 255 and green and blue to 0 creates an intense red.

Selecting colors with convenient numbers can save effort. For example, it's common to see the parameters (0, 0, 255) used for blue and (0, 255, 0) for green. These combinations are often responsible for the garish coloring associated with technical images produced on the computer. They seem extreme and unnatural because they don't account for the human eye's ability to distinguish subtle values. Colors that appeal to our eyes are usually not convenient numbers. Rather than picking numbers like 0 and 255, try using a color selector and choosing colors. Processing's color selector is opened from the Tools menu. Colors are selected by clicking a location on the color field or entering numbers directly. For example, in the figure on the facing page, the current blue selected is defined by an R value of 35, a G value of 211, and a B value of 229. These numbers can be used to recreate the chosen color in your code.

Setting colors

In Processing, colors are defined by the parameters to the `background()`, `fill()`, and `stroke()` functions:

```
background(value1, value2, value3)
fill(value1, value2, value3)
fill(value1, value2, value3, alpha)
stroke(value1, value2, value3)
stroke(value1, value2, value3, alpha)
```

By default, the *value1* parameter defines the red color component, *value2* the green component, and *value3* the blue. The optional alpha parameter to `fill()` or `stroke()` defines the transparency. The *alpha* parameter value 255 means the color is entirely opaque, and the value 0 means it's entirely transparent (it won't be visible).



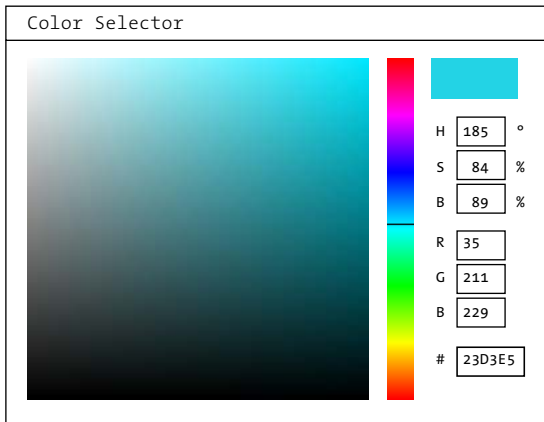
```
background(242, 204, 47);
```

9-01



```
background(174, 221, 60);
```

9-02



Color Selector

Drag the cursor inside the window or input numbers to select a color. The large square area determines the saturation and brightness, and the thin vertical strip determines the hue. The numeric value of the selected color is displayed in HSB, RGB, and hexadecimal notation.



```
background(129, 130, 87);
noStroke();
fill(174, 221, 60);
rect(17, 17, 66, 66);
```

9-03



```
background(129, 130, 87);
noFill();
strokeWeight(4);
stroke(174, 221, 60);
rect(19, 19, 62, 62);
```

9-04



```
background(116, 193, 206);
noStroke();
fill(129, 130, 87, 102); // More transparent
rect(20, 20, 30, 60);
fill(129, 130, 87, 204); // Less transparent
rect(50, 20, 30, 60);
```

9-05



```
background(116, 193, 206);
int x = 0;
noStroke();
for (int i = 51; i <= 255; i += 51) {
  fill(129, 130, 87, i);
  rect(x, 20, 20, 60);
  x += 20;
}
```

9-06



```
background(56, 90, 94);
smooth();
strokeWeight(12);
stroke(242, 204, 47, 102); // More transparency
line(30, 20, 50, 80);
stroke(242, 204, 47, 204); // Less transparency
line(50, 20, 70, 80);
```

9-07



```
background(56, 90, 94);
smooth();
int x = 0;
strokeWeight(12);
for (int i = 51; i <= 255; i += 51) {
  stroke(242, 204, 47, i);
  line(x, 20, x+20, 80);
  x += 20;
}
```

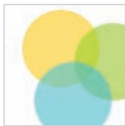
9-08

Transparency can be used to create new colors by overlapping shapes. The colors originating from overlaps depend on the order in which the shapes are drawn.



```
background(0);
noStroke();
smooth();
fill(242, 204, 47, 160); // Yellow
ellipse(47, 36, 64, 64);
fill(174, 221, 60, 160); // Green
ellipse(90, 47, 64, 64);
fill(116, 193, 206, 160); // Blue
ellipse(57, 79, 64, 64);
```

9-09



```
background(255);
noStroke();
smooth();
fill(242, 204, 47, 160); // Yellow
ellipse(47, 36, 64, 64);
fill(174, 221, 60, 160); // Green
ellipse(90, 47, 64, 64);
fill(116, 193, 206, 160); // Blue
ellipse(57, 79, 64, 64);
```

9-10

Color data

The `color` data type is used to store colors in a program, and the `color()` function is used to assign a `color` variable. The `color()` function can create `gray` values, `gray` values with transparency, `color` values, and `color` values with transparency. Variables of the `color` data type can store all of these configurations:

```
color(gray)
color(gray, alpha)
color(value1, value2, value3)
color(value1, value2, value3, alpha)
```

The parameters of the `color()` function define a color. The `gray` parameter used alone or with `alpha` defines tones ranging from white to black. The `alpha` parameter defines transparency with values ranging from 0 (transparent) to 255 (opaque). The `value1`, `value2`, and `value3` parameters define values for the different components. Variables of the `color` data type are defined and assigned in the same way as the `int` and `float` data types discussed in Data 1 (p. 37).

```
color c1 = color(51);           // Creates gray                      9-11
color c2 = color(51, 204);      // Creates gray with transparency
color c3 = color(51, 102, 153); // Creates blue
color c4 = color(51, 102, 153, 51); // Creates blue with transparency
```

After a `color` variable has been defined, it can be used as the parameter to the `background()`, `fill()`, and `stroke()` functions.



```
color ruby = color(211, 24, 24, 160);
color pink = color(237, 159, 176);
background(pink);
noStroke();
fill(ruby);
rect(35, 0, 20, 100);
```

9-12

RGB, HSB

Processing uses the RGB color model as its default for working with color, but the HSB specification can be used instead to define colors in terms of their hue, saturation, and brightness. The hue of a color is what most people normally think of as the color name: yellow, red, blue, orange, green, violet. A pure hue is an undiluted color at its most intense. The saturation is the degree of purity in a color. It is the continuum from the undiluted, pure hue to its most diluted and dull. The brightness of a color is its relation to light and dark.

	RGB			HSB			HEX
	255	0	0	360	100	100	#FF0000
	252	9	45	351	96	99	#F0A2E
	249	16	85	342	93	98	#F91157
	249	23	126	332	90	98	#F91881
	246	31	160	323	87	97	#F720A4
	244	38	192	314	84	96	#F427C4
	244	45	226	304	81	96	#F42EE7
	226	51	237	295	78	95	#E235F2
	196	58	237	285	75	95	#C43CF2
	171	67	234	276	71	94	#AB45EF
	148	73	232	267	68	93	#944BED
	126	81	232	257	65	93	#7E53ED
	108	87	229	248	62	92	#6C59EA
	95	95	227	239	59	91	#5F61E8
	102	122	227	229	56	91	#667DE8
	107	145	224	220	53	90	#6B94E5
	114	168	224	210	50	90	#72ACE5
	122	186	221	201	46	89	#7ABEE2
	127	200	219	192	43	88	#7FCDE0
	134	216	219	182	40	88	#86DDE0
	139	216	207	173	37	87	#8BDD04
	144	214	195	164	34	86	#90DBC7
	151	214	185	154	31	86	#97DBBD
	156	211	177	145	28	85	#9CD8B5
	162	211	172	135	25	85	#A2D8B0
	169	209	169	126	21	84	#A9D6AD
	175	206	169	117	18	83	#AFD3AD
	185	206	175	107	15	83	#BAD3B3
	192	204	180	98	12	82	#C1D1B8
	197	201	183	89	9	81	#C5CEBB
	202	201	190	79	6	81	#CACEC2
	202	200	193	70	3	80	#CACCC5

Color by numbers

Every color within a program is set by numbers, and there are more than 16 million colors to choose from. This diagram presents a few colors and their corresponding numbers for the RGB and HSB color models. The RGB column is in relation to `colorMode(RGB, 255)` and the HSB column is in relation to `colorMode(HSB, 360, 100, 100)`.

The `colorMode()` function sets the color space for a program:

```
colorMode(mode)
colorMode(mode, range)
colorMode(mode, range1, range2, range3)
```

The parameters to `colorMode()` change the way Processing interprets color data. The *mode* parameter can be either RGB or HSB. The range parameters allow Processing to use different values than the default of 0 to 255. A range of values frequently used in computer graphics is between 0.0 and 1.0. Either a single range parameter sets the range for all the color components, or the *range1*, *range2*, and *range3* parameters set the range for each—either red, green, blue or hue, saturation, brightness, depending on the value of the *mode* parameter.

```
// Set the range for the red, green, and blue values from 0.0 to 1.0 9-13
colorMode(RGB, 1.0);
```

A useful setting for HSB mode is to set the *range1*, *range2*, and *range3* parameters respectively to 360, 100, and 100. The hue values from 0 to 360 are the degrees around the color wheel, and the saturation and brightness values from 0 to 100 are percentages. This setting matches the values used in many color selectors and therefore makes it easy to transfer color data between other programs and Processing:

```
// Set the range for the hue to values from 0 to 360 and the 9-14
// saturation and brightness to values between 0 and 100
colorMode(HSB, 360, 100, 100);
```

The following examples reveal the differences between hue, saturation, and brightness.



```
// Change the hue, saturation and brightness constant 9-15
colorMode(HSB);
for (int i = 0; i < 100; i++) {
  stroke(i*2.5, 255, 255);
  line(i, 0, i, 100);
}
```



```
// Change the saturation, hue and brightness constant 9-16
colorMode(HSB);
for (int i = 0; i < 100; i++) {
  stroke(132, i*2.5, 204);
  line(i, 0, i, 100);
}
```




```
// Change the brightness, hue and saturation constant
colorMode(HSB);
for (int i = 0; i < 100; i++) {
  stroke(132, 108, i*2.5);
  line(i, 0, i, 100);
}
```

9-17



```
// Change the saturation and brightness, hue constant
colorMode(HSB);
for (int i = 0; i < 100; i++) {
  for (int j = 0; j < 100; j++) {
    stroke(132, j*2.5, i*2.5);
    point(i, j);
  }
}
```

9-18

It's easy to make smooth transitions between colors by changing the values used for *color()*, *fill()*, and *stroke()*. The HSB model has an enormous advantages over the RGB model when working with code because it's more intuitive. Changing the values of the red, green, and blue components often has unexpected results, while estimating the results of changes to hue, saturation, and brightness follows a more logical path. The following examples show a transition from green to blue. The first example makes this transition using the RGB model. It requires calculating all three color values, and the saturation of the color unexpectedly changes in the middle. The second example makes the transition using the HSB model. Only one number needs to be altered, and the hue changes smoothly and independently from the other color properties.



```
// Shift from blue to green in RGB mode
colorMode(RGB);
for (int i = 0; i < 100; i++) {
  float r = 61 + (i*0.92);
  float g = 156 + (i*0.48);
  float b = 204 - (i*1.43);
  stroke(r, g, b);
  line(i, 0, i, 100);
}
```

9-19



```
// Shift from blue to green in HSB mode
colorMode(HSB, 360, 100, 100);
for (int i = 0; i < 100; i++) {
  float newHue = 200 - (i*1.2);
  stroke(newHue, 70, 80);
  line(i, 0, i, 100);
}
```

9-20

Hexadecimal

Hexadecimal (hex) notation is an alternative notation for defining color. This method is popular with designers working on the Web because standards such as HyperText Markup Language (HTML) and Cascading Style Sheets (CSS) use this notation. Hex notation for color encodes each of the numbers from 0 to 255 into a two-digit value using the numbers 0 through 9 and the letters A through F. In this way three RGB values from 0 to 255 can be written as a single six-digit hex value. A few sample conversions demonstrate this notation:

RGB	Hex
255, 255, 255	#FFFFFF
0, 0, 0	#000000
102, 153, 204	#6699CC
195, 244, 59	#C3F43B
116, 206, 206	#74CECE

Converting color values from RGB to hex notation is not intuitive. Most often, the value is taken from a color selector. For instance, you can copy and paste a hex value from Processing's color selector into your code. When using color values encoded in hex notation, you must place a # before the value to distinguish it within the code.



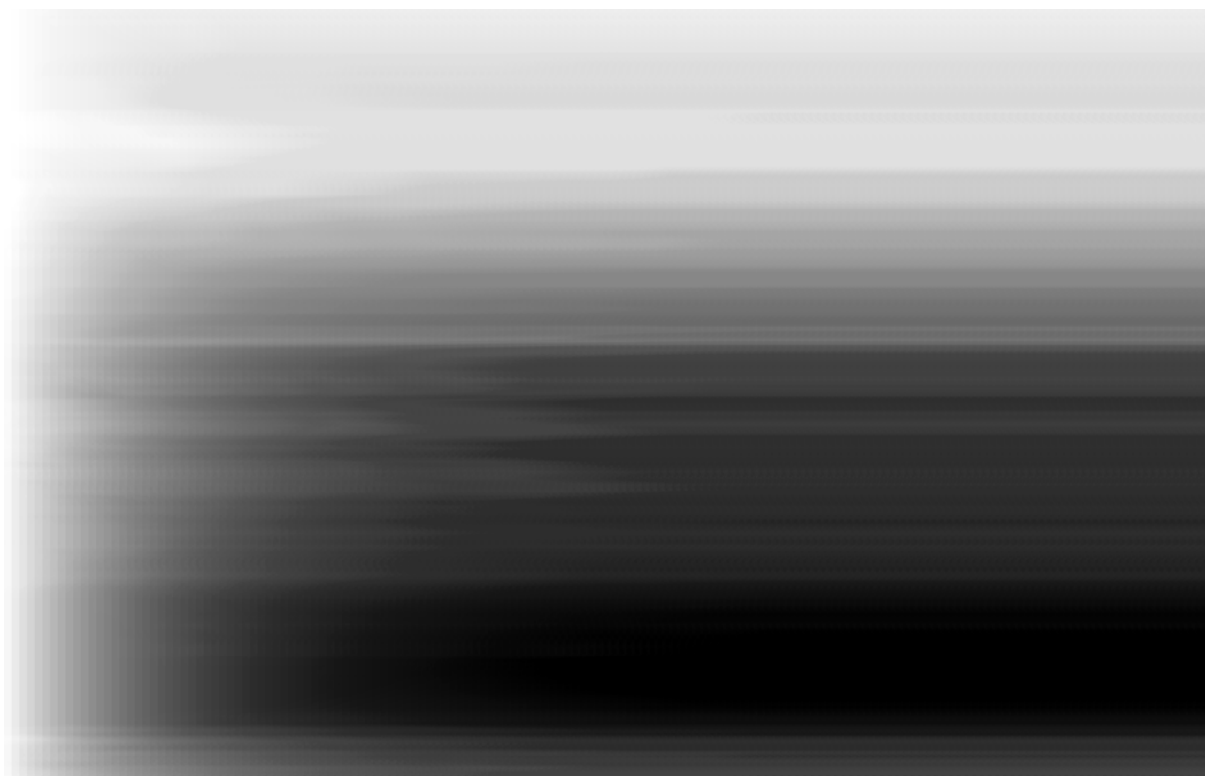
```
// Code 9-03 rewritten using hex numbers  
background(#818257);  
noStroke();  
fill(#AEDD3C);  
rect(17, 17, 66, 66);
```

9-21

There's more information about hex notation in Appendix D (p. 669).

Exercises

1. Explore a wide range of color combinations within one composition.
2. Use HSB color and a *for* structure to design a gradient between two colors.
3. Redraw your composition from exercise 1 using hexadecimal color values.



Color 2: Components

This unit introduces functions for reading the components of a color and discusses techniques for creating dynamic color palettes.





















Syntax introduced:

`red()`, `blue()`, `green()`, `alpha()`, `hue()`, `saturation()`, `brightness()`

Colors are stored in software as numbers. Each color is defined by its component elements. When color is defined by RGB values, there are three numbers that store the red, green, and blue components and an optional fourth number that stores a transparency value. When working with HSB values, three numbers store the hue, saturation, and brightness values and a fourth denotes the transparency. The visible color is a combination of these components. Adjusting the individual color properties in isolation from the others is a useful technique for dynamically changing a single color value or the entire palette for a program.

Extracting color

In Processing, the `color` data type is a single number that stores the individual components of a color. This value combines the red, green, blue, and alpha (transparency) components. Behind the scenes, this value is actually an `int`, and can be used interchangeably with an `int` variable anywhere in a program. The `color` data type stores the components of the color as a series of values from 0 to 255 embedded into this larger number. We can look at an abstracted view of a color with this table:

Red	Green	Blue	Alpha		Color
				→	
64	124	188	255		
				→	
151	186	66	255		
				→	
214	124	43	255		
				→	
214	124	43	126		

The `red()`, `green()`, and `blue()` functions are used for reading the components of a color. The `red()` function extracts the red component, the `green()` function extracts the green component, and the `blue()` function extracts the blue component.

```

color c1 = color(0, 126, 255);           // Create a new color           38-01
float r = red(c1);                       // Assign 0.0 to r
float g = green(c1);                     // Assign 126.0 to g
float b = blue(c1);                       // Assign 255.0 to b
println(r + ", " + g + ", " + b);        // Prints "0.0, 126.0, 255.0"
color c2 = color(102);                   // Create a new gray value
float r2 = red(c2);                       // Assign 102.0 to r2
float g2 = green(c2);                     // Assign 102.0 to g2
float b2 = blue(c2);                     // Assign 102.0 to b2
println(r2 + ", " + g2 + ", " + b2);     // Prints "102.0, 102.0, 102.0"

```

The `alpha()` function reads the alpha value of the color. Remember, a fourth value added to the `color()` function sets the transparency value for this color. If no alpha value is set, the default 255 is used.

```

color c = color(0, 51, 102);             // Create a new color
color g = color(0, 126, 255, 220);       // Create a new color           38-02
float a = alpha(c);                       // Assign 255.0 to a
float b = alpha(g);                       // Assign 220.0 to b
println(a + ", " + b);                   // Prints "255.0, 220.0"

```

The `hue()`, `saturation()`, and `brightness()` functions work like `red()`, `green()`, and `blue()`, but return different components of the color. It makes sense to switch to the HSB color model when using these functions, but sometimes you will want these components while in the default RGB color mode.

```

colorMode(HSB, 360, 100, 100);           // Set color mode to HSB
color c = color(210, 100, 40);           // Create a new color           38-03
float h = hue(c);                         // Assign 210.0 to h
float s = saturation(c);                   // Assign 100.0 to s
float b = brightness(c);                   // Assign 40.0 to b
println(h + ", " + s + ", " + b);         // Prints "210.0, 100.0, 40.0"

```

```

color c = color(217, 41, 117);           // Create a new color
float r = red(c);                         // Assign 217.0 to r           38-04
float h = hue(c);                         // Assign 236.64774 to h
println(r + ", " + h);                     // Prints "217.0, 236.64774"

```

The values from all of these functions are scaled based on the current color mode settings. If the range for color values is changed with `colorMode()`, the values returned will be scaled within the new range.

```
colorMode(RGB, 1.0);           // Sets color mode to HSB
color c = color(0.2, 0.8, 1.0); // Creates a new color
float r = red(c);              // Assign 0.2 to r
float h = hue(c);              // Assign 0.5416667 to h
println(r + ", " + h);         // Prints "0.2, 0.5416667"
```

38-05

The values returned from these color functions are always floating-point values, therefore you'll receive an error if you try to assign the result to an integer value. If you need the result to be an integer, you can simply convert the value using the `int()` function (p. 107).

```
color c = color(118, 22, 24); // Create a new color
int r1 = red(c);              // ERROR! red() returns a float
float r2 = red(c);            // Assign 118.0 to r2
int r3 = int(red(c));         // Assign 118 to r3
```

38-06

As described in Image 3 (p. 321), these functions make it possible to read the individual color components of the pixels in the display window. In the following examples, the `get()` function is used to access the color at the current cursor position. The components of these colors are extracted and used to set the drawing properties.



```
// Sets the stroke color of the lines to the
// red component of the pixel below their intersection
```

38-07



```
void setup() {
  size(100, 100);
  smooth();
  fill(204, 0, 0);
}
```



```
void draw() {
  background(0);
  noStroke();
  ellipse(66, 46, 80, 80);
  color c = get(mouseX, mouseY);
  float r = red(c); // Extract red component
  stroke(255-r);    // Set the stroke based on red value
  line(mouseX, 0, mouseX, height);
  line(0, mouseY, width, mouseY);
}
```



// Simulates one pixel of a flat-panel display

38-08



PImage wall;

```
void setup() {
  size(100, 100);
  wall = loadImage("veg.jpg");
  stroke(255);
}
```



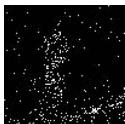
```
void draw() {
  background(wall);
  color c = get(mouseX, mouseY);
  float r = red(c);    // Extract red
  float g = green(c);  // Extract green
  float b = blue(c);   // Extract blue
  fill(r, 0, 0);
  rect(32, 20, 12, 60); // Red component
  fill(0, g, 0);
  rect(44, 20, 12, 60); // Green component
  fill(0, 0, b);
  rect(56, 20, 12, 60); // Blue component
}
```

Values extracted with the `red()`, `green()`, and `blue()` functions can be used in many different ways. For instance, the numbers can be used to control aspects of motion or the flow of the program. In the following example, the brightness of pixels in an image controls the speed of 400 points moving across the screen. Each point moves across the screen from left to right. The pixel value in the image with the same coordinate as a point is read and used to set the speed at which the point moves. Each point moves slowly through dark areas and quickly through lighter areas. Run the code and try a different photo to see how the same program can be used to create different patterns of motion.



```
int num = 400;
float[] x = new float[num];
float[] y = new float[num];
PImage img;
```

38-09



```
void setup() {
  size(100, 100);
  img = loadImage("standing-alt.jpg");
  for (int i = 0; i < num; i++) {
    x[i] = random(width);
    y[i] = random(height);
  }
```

```

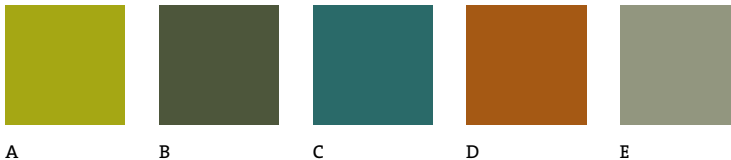
    }
    stroke(255);
}

void draw() {
  background(0);
  for (int i = 0; i < num; i++) {
    color c = img.get(int(x[i]), int(y[i]));
    float b = brightness(c) / 255.0;
    float speed = pow(b, 2) + 0.05;
    x[i] += speed;
    if (x[i] > width) {
      x[i] = 0;
      y[i] = random(height);
    }
    point(x[i], y[i]);
  }
}

```

Dynamic color palettes

One of the most important concepts in working with color is relativity. When one color is positioned next to another, they both appear to change. If a color is to appear the same in a new juxtaposition, it often must be physically different (defined with different numbers). This is important to consider when working with color in software, since elements are often moving and changing colors. For example, placing these five colors ...



... in a different order changes their appearance:



The phenomenon of color relativity can be extended in software by linking colors' relations and making them change dynamically in response to input from the mouse. In the following example, four colors are used. The colors stored in the variables `olive`

and gray remain the same, while the values for yellow and orange change in relation to mouseY and therefore shift as the cursor moves up and down.



```
color olive, gray;
```

```
void setup() {
  size(100, 100);
  colorMode(HSB, 360, 100, 100, 100);
  noStroke();
  smooth();
  olive = color(75, 61, 59);
  gray = color(30, 17, 42);
}
```

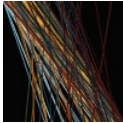
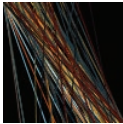
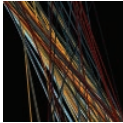
```
void draw() {
  float y = mouseY / float(height);
  background(gray);
  fill(olive);
  quad(70 + y*6, 0, 100, 0, 100, 100, 30 - y*6, 100);
  color yellow = color(48 + y*20, 100, 88 - y*20);
  fill(yellow);
  ellipse(50, 45 + y*10, 60, 60);
  color orange = color(29, 100, 83 - y*10);
  fill(orange);
  ellipse(54, 42 + y*16, 24, 24);
}
```

38-10

A good technique for creating subtle and complex color palettes with software is to use colors directly from images. Image can be loaded into the software and their colors read using the `get()` function. For the examples in the rest of this section, the 100 color values from a 10×10 pixel image are used to set the fill and stroke colors of shapes. To show the different values more clearly, the image has been enlarged:



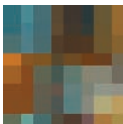
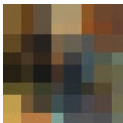
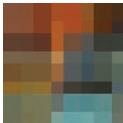
Depending on your goals, you can load a photographic image or one that has been constructed pixel by pixel. An image of any dimension can be loaded and used as a color palette. Sometimes it's appropriate to use only a few colors, and other times hundreds of unique colors might be desired.



```
PImage img;
```

38-11

```
void setup() {  
  size(100, 100);  
  smooth();  
  frameRate(0.5);  
  img = loadImage("palette10x10.jpg");  
}  
  
void draw() {  
  background(0);  
  for (int x = 0; x < img.width; x++) {  
    for (int y = 0; y < img.height; y++) {  
      float xpos1 = random(x*10);  
      float xpos2 = width - random(y*10);  
      color c = img.get(x, y);  
      stroke(c);  
      line(xpos1, 0, xpos2, height);  
    }  
  }  
}
```



```
PImage img;
```

38-12

```
void setup() {  
  size(100, 100);  
  noStroke();  
  img = loadImage("palette10x10.jpg");  
}  
  
void draw() {  
  int ix = int(random(img.width));  
  int iy = int(random(img.height));  
  color c = img.get(ix, iy);  
  fill(c, 102);  
  int xgrid = int(random(-2, 5)) * 25;  
  int ygrid = int(random(-2, 5)) * 25;  
  rect(xgrid, ygrid, 40, 40);  
}
```

Loading the colors from the image into an array opens more possibilities. Once the colors are in an array, they can be easily reordered or shifted. In the following example, the color values from the image are loaded sequentially into an array and then reordered according to their brightness. The `sortColors()` function takes an array of colors as an input, puts them in order from dark to light, and then returns the sorted colors. As it counts from 0 to 255, it puts all the colors with the current value from the unsorted array into the new array.



Original array



Array sorted by brightness

The following example uses the values of the sorted array elements to determine the thickness and center point of the line pairs drawn to the display window. Each pair of lines is spaced evenly at ten-pixel intervals, and a random value is used to access a color from the `imageColors[]` array. Because the colors in the array are sorted, line 24 ensures that the thin lines are bright and the thick lines are dark, regardless of their hue and saturation.



```
PImage img;
color[] imageColors;

void setup() {
  size(100, 100);
  frameRate(0.5);
  smooth();
  noFill();
  img = loadImage("palette10x10.jpg");
  imageColors = new color[img.width*img.height];
  for (int y = 0; y < img.height; y++) {
    for (int x = 0; x < img.width; x++) {
      imageColors[y*img.height + x] = img.get(x, y);
    }
  }
  imageColors = sortColors(imageColors);
}

void draw() {
  background(255);
  for (int x = 10; x < width; x += 10) {
    int r = int(random(imageColors.length));
    float thick = ((100-r) / 4.0) + 1.0;
```

38-13

```

        stroke(imageColors[r]);
        strokeWeight(thick);
        line(x, height, x, height-r+thick);
        line(x, 0, x, height-r-thick);
    }
}

color[] sortColors(color[] colors) {
    color[] sorted = new color[colors.length];
    int num = 0;
    for (int i = 0; i <= 255; i++) {
        for (int j = 0; j < colors.length; j++) {
            if (int(brightness(colors[j])) == i) {
                sorted[num] = colors[j];
                num++;
            }
        }
    }
    return sorted;
}

```

Exercises

1. Write a program to print the red, green, and blue values of every pixel in an image to the console.
2. Design a composition that changes based on the `mouseX` value. Make the color for each element of the composition also change in relation to this variable.
3. Load an image and use its colors to set the palette for a composition.

Bit, Binary, Hex

Bit

A bit (binary digit) is the most basic information unit in computing. It's often thought of as a 1 or 0, but a bit has no numeric meaning. It's simply a way to distinguish between two mutually exclusive states. Bits may be stored as holes punched in a card, a positive or negative magnetic charge on a floppy disk, or an indent in the surface of a compact disk. The amazing innovation of binary notation is the ability to encode many types of data and logic with only two different states. This was made possible by George Boole's contributions to logic in the mid-nineteenth century and Claude Shannon's development of information theory in the 1930s. The information that comprises images, video, text, and software is all encoded into binary notation and later decoded as colors, shapes, and words that we are able to understand. Bits are grouped together in units of 8 called bytes. Storage on computers is measured in these units. For example, a kilobyte (K, KB, kB, Kbyte) is 1024 bytes, a megabyte (MB) is 1,048,576 bytes, and a gigabyte (GB, Gbyte) is 1,073,741,824 bytes.¹

Binary

The binary number system, also called base-2, represents numbers as sequences of 1s and 0s. This is different from the more common decimal representation, also called base-10. Here we can compare the powers of 10 and the powers of 2:

Base-10	10^0	10^1	10^2	10^3	10^4	10^5
	1	10	100	1000	10000	100000
Base-2	2^0	2^1	2^2	2^3	2^4	2^5
	1	2	4	8	16	32

When using a computer, it's clear that many frequently used numbers are a result of base-2 notation. For example, colors are specified in values from 0 to 255 (2^8 , the number of unique values for one byte), and screens are often 1024 pixels wide (2^{10}).

In base-10 numbers, each digit is multiplied by the power of 10 associated with its position. For example, the number 243 is expressed as follows:

$$\begin{aligned}
 200 &+ 40 &+ 3 &= 243 \\
 2*100 &+ 4*10 &+ 3*1 &= 243 \\
 2*10^2 &+ 4*10^1 &+ 3*10^0 &= 243
 \end{aligned}$$

Base-2 numbers work the same way, but the digits are multiplied by the powers of 2. Every whole number can be made by adding values that are powers of two. The following example breaks down the binary equivalent of the decimal number 23 (16+4+2+1) which is 10111:

$$\begin{aligned}
 16 &+ 0 &+ 4 &+ 2 &+ 1 &= 23 \\
 1*16 &+ 0*8 &+ 1*4 &+ 1*2 &+ 1*1 &= 23 \\
 1*2^5 &+ 0*2^4 &+ 1*2^2 &+ 1*2^1 &+ 1*2^0 &= 23
 \end{aligned}$$

Each file format specifies how information is encoded into binary notation and software is used to decode the information according to the standards for each format. For example, using the ASCII standard (p. 664) for text, the word Process is encoded as the numeric and binary sequence like this:

Character	P	r	o	c	e	s	s
ASCII	80	114	111	99	101	115	115
Binary	01010000	01110010	01101111	01100011	01100101	01110011	01110011

Bitwise operations

The integer and floating-point representations of numbers are operated on with arithmetic operators such as + and *. The binary representations of numbers have different operators. Bitwise operators & (bitwise AND) and | (bitwise OR) are used for comparing binary representations. The bitwise operators >> and << are used to shift bits left and right.

The bitwise AND operator compares each corresponding bit according to these rules:

Expression	Evaluation
1 & 1	1
1 & 0	0
0 & 0	0

A larger calculation follows:

$$\begin{array}{r}
 11010110 \\
 \& 01011100 \\
 \hline
 01010100
 \end{array}$$

The bitwise OR operator compares each corresponding bit according to these rules.

<i>Expression</i>	<i>Evaluation</i>
1 1	1
1 0	1
0 0	0

A larger calculation follows:

```
11010110
| 01011100
-----
11011110
```

The bitwise operators >> and << shift bits left and right.

```
int a = 205; // In binary: 00000000000000000000000011001101
int b = 45;  // In binary: 0000000000000000000000000101101
a = a << 24; // Converts to 11001101000000000000000000000000
b = b << 8;  // Converts to 0000000000000000000010110100000000
```

D-01

Hex

Hexadecimal notation encodes an entire 8-digit byte with just two characters, one character for each *nibble* (4 bits, or half a byte). Because there are only 16 possible byte configurations for a nibble, each can be encoded with the following 16 distinct alphanumeric characters.

0000	0	0100	4	1000	8	1100	C
0001	1	0101	5	1001	9	1101	D
0010	2	0110	6	1010	A	1110	E
0011	3	0111	7	1011	B	1111	F

For example, the binary sequence ...

```
01010000 01110010 01101111 01100011 01100101 01110011 01110011
```

... is reduced to this hexadecimal encoding:

```
50 72 6F 63 65 73 73
```

Hex notation is an established way to define color within software and on the Web. For example, the three decimal RGB color value 255, 204, 51 is converted to FFCC33 in

hexadecimal notation. In Processing, a # sign in front of six digits denotes a web color. To use hexadecimal notation for other uses besides color, place 0x (the number zero followed by a lower-case x) in front of the digits.

Notes

1. There are two definitions for kilobyte, megabyte, and gigabyte. The alternative quantities are respectively, 1,000 (one thousand), 1,000,000 (one million), and 1,000,000,000 (one billion).

Optimization

Optimization is making changes to a program so that it will run faster. This can provide tremendous benefit by increasing the number of frames displayed per second or by allowing more to be drawn to the screen each frame. Increasing the speed can also make a program more responsive to mouse and keyboard input.

Code should usually not be optimized until a late stage in a program's development. Energy diverted to optimization detracts from refining the concept and aesthetic considerations of the software. Optimization can be very rewarding because of increased performance, but such technical details should not be allowed to distract you from the ideas. There are a few important heuristics to guide the process:

Work slowly and carefully. It's easy to introduce new bugs when optimizing, so work with small pieces of code at a time. Always keep the original version of the code. You may want to comment out the old version of a function and keep it present in your program while you work on its optimization.

Optimize the code that's used most. The majority of code in a program gets used very little, so make sure that you're focusing on a section that needs work. Advanced programmers can use a *profiler* for this task—a tool that identifies the amount of time being spent in different sections of code. Profilers are too specific to be covered here, but books and online references cover profiling Java code, and this methodology can be applied to Processing programs.

If the optimization doesn't help, revert to the original code. Lots of things seem like they'll improve performance but actually don't (or they don't improve things as much as hoped). The "optimized" version of the code will usually be more difficult to read—so if the benefits aren't sufficient, the clearer version is better.

There are many techniques to optimize programs; some that are particularly relevant to Processing sketches are listed below.

Bit-shifting color data

The `red()`, `green()`, `blue()`, and `alpha()` functions are easy to use and understand, but because they take the `colorMode()` setting into account, using them is much slower than making direct operations on the numbers. With the default color mode, the same numerical results can be achieved with greater speed by using the `>>` (right shift) operator to isolate the components and then use the bit mask `0xFF` to remove any unwanted data. These operators are explained in Appendix D (p. 669). The following example shows how to shift color data to isolate each component.

E-01

AAAAAAAAARRRRRRRRGGGGGGGGBBBBBBBB

OOOOOOOOOOOOOOOOOOOOOOOAAAAAAAAARRRRRRRRR

$$\begin{array}{r} \text{0000000000000000AAAAAAARRRRRRR} \\ \& \text{00000000000000000000000011111111} \\ \hline \text{000000000000000000000000RRRRRRR} \end{array}$$

```
int a = 255;
int r = 102;
int g = 51;
int b = 255;
color c = (a << 24) | (r << 16) | (g << 8) | b;
```

```
color c = 0xFF000000 | (r << 16) | (g << 8) | b;
```

Bit shifting is much faster than using the `color()` method, because it ignores the `colorMode()` setting. For the same reason, specifying colors using hex notation (e.g., `#FFCC00`) has zero overhead.

Avoid creating objects in draw()

Creating an object slows a program down. When possible, create the objects within `setup()` so they are created only once within the program. For example, load all images and create objects within `setup()`. The following two examples show common misunderstandings about creating objects that slow programs down.

```
// AVOID loading an image within draw(), it is slow
void draw() {
    PImage img = loadImage("tower.jpg");
    image(img, 0, 0);
}
```

E-03

```
// AVOID creating an array inside draw(), it is slow
void draw() {
    int[] values = new int[200];
    // Do something with the array here
}
```

E-04

In this case, the array will be re-created and destroyed on each trip through the `draw()` method, which is extremely wasteful. The programs 43-07 (p. 404) and 44-05 (p. 417) show faster ways of creating objects.

Using the pixels[] array

The `get()` and `set()` functions are easy to use, but they are not as fast as accessing and setting the pixels of an image directly through the `pixels[]` array (p. 356). The following examples show four different ways of accessing the data within the `pixels[]` array, each faster than the previous one.

```
// Converts (x,y) coordinates into a position in the pixels[] array
loadPixels();
for (int y = 0; y < height; y++) {
    for (int x = 0; x < width; x++) {
        pixels[y*height + x] = color(102);
    }
}
updatePixels();
```

E-05

```
// Replaces the multiplication y*height with an addition
int offset = 0;
loadPixels();
for (int y = 0; y < height; y++) {
```

E-06

```

    for (int x = 0; x < width; x++) {
        pixels[offset + x] = color(102);
    }
    offset += width; // Avoids the multiply
}
updatePixels();

```

E-06
cont.

```

// Avoid the calculation y*height+width
int index = 0;
loadPixels();
for (int y = 0; y < height; y++) {
    for (int x = 0; x < width; x++) {
        pixels[index++] = color(102);
    }
}
updatePixels();

```

E-07

```

// Avoids (x,y) coordinates
int wh = width*height;
loadPixels();
for (int index = 0; index < wh; index++) {
    pixels[index] = color(102);
}
updatePixels();

```

E-08

```

// Only calculate the color once
int wh = width*height;
color c = color(102);
loadPixels();
for (int index = 0; index < wh; index++) {
    pixels[index] = c;
}
updatePixels();

```

E-09

When manipulating pixels[, use the loadPixels() and updatePixels() functions only once within draw(). When possible, use color() outside of loops. It is not very fast because it must take into account the current color mode.

Tips for working with arrays

Adding one value at a time to an array is slower than doubling the size of the array when it's full. If a program will be continually adding data to the end of an array, use the `expand()` function once each time the array fills up in place of running `append()`

many times. Code 33-19 (p. 309) shows how to manage a growing array with `expand()`.

The `arraycopy()` function is the fastest way of copying data from one array to another. Copying the data from one array to another inside a `for` structure is much slower when copying data from large arrays. The `arraycopy()` function is demonstrated in code 33-20 (p. 310). Arrays are also much faster (sometimes 2×) than the Java classes `ArrayList` and `Vector`.

Avoid repeating calculations

If the same calculation is made more than once, it's faster to make the calculation once and store it in a variable. Instead of writing ...

```
float x = (width/2) * 4;
float y = (width/2) * 8;
```

... save the result of the division in a variable and substitute it for the calculation:

```
float half = width/2;
float x = half * 4;
float y = half * 8;
```

Multiplications and divisions from inside a `for` structure can slow a program down significantly, especially if there are more than 10,000 iterations. When possible, make these calculations outside of the structure. This is demonstrated above in code E-08.

Because of the way computers make calculations, addition is faster than multiplication and multiplication is faster than division. Multiplication can often be converted to addition by restructuring the program. For example, compare the difference in run time between code E-05 and code E-06.

Lookup tables

The idea behind a lookup table is that it is faster to make reference to a value stored within a data structure than to make the calculation. One example is making calculations for `sin()` and `cos()` at each frame. These numbers can be generated once within `setup()` and stored within an array so they may be quickly retrieved. The following example shows how it's done.

```
int res = 16;                                // Number of data elements
float[] x = new float[res];                  // Create x-coordinate array
float[] y = new float[res];                  // Create y-coordinate array

void setup() {
    size(100, 100);
```

E-10

```

    for (int i = 0; i < res; i++) {
        x[i] = cos(PI/res * i);           // Sets x-coordinates
        y[i] = sin(PI/res * i);           // Sets y-coordinates
    }
}

void draw() {
    for (int i = 0; i < res; i++) {       // Access each point
        point(50 + x[i]*40, 50 + y[i]*40); // Draws point on a curve
    }
}

```

You can change the resolution of the values by altering the length of the array.

Optimizers beware!

Optimized code can sometimes be more difficult to read. When deciding whether to optimize, balance the need for speed against the value of legibility. For example, in the bit-shifting example presented above, the expression

red(color)

is more clear than its optimized equivalent:

(color >> 16) & 0xFF

The name of the `red()` function clearly states its purpose, whereas the bit shift and mask are cryptic to everyone except those familiar with the technique. The confusion can be alleviated with comments, but always consider whether the optimization is more important than the simplicity of your code..