

Chapter 2. Starting to Code

To get the most out of this book, you need to do more than just read the words. You need to experiment and practice. You can't learn to code just by reading about it—you need to do it. To get started, download p5.js and make your first sketch.

Environment

First, you'll need to get a code editor. A code editor is similar to a text editor (like Notepad or Notes), except it has special functionality for editing code instead of plain text. You can use any code editor you like; we recommend [Atom](#) and [Brackets](#), both of which can be downloaded online.

There is also an official p5.js editor in development. If you would like to use it, you can download it by visiting <http://p5js.org/download> and selecting the button under “Editor.” If you are using the p5.js editor, you can skip ahead to [“Your First Program”](#).

Download and File Setup

Start by visiting <http://p5js.org/download> and selecting “p5.js complete.” Double-click the *.zip* file that downloads, and drag the folder inside to a location on your hard disk. It could be *Program Files* or *Documents* or simply the desktop, but the important thing is for the *p5* folder to be pulled out of that *.zip* file.

The *p5* folder contains an example project that you can begin working from. Open your code editor. Next, you'll want to open the folder named *empty-example* in your code editor. In most code editors, you can do this by going to the File menu in your editor and choosing Open..., then selecting the folder *empty-example*. You're now all set up and ready to begin your first program!

Your First Program

When you open the *empty-example* folder, you will likely see a sidebar with the folder name at the top, and a list of the files contained in the folder directly below. If you click each of these files, you will see the contents of the file appear in the main area.

A p5.js sketch is made from a few different languages used together. *HTML* (HyperText Markup Language) provides the backbone, linking all the other elements together in a page. JavaScript (and the p5.js library) enable you to create interactive graphics that display on your HTML page. Sometimes *CSS* (Cascading Style Sheets) are used to further style elements on the HTML page, but we won't cover that in this book.

If you look at the *index.html* file, you'll notice that there is some HTML code there. This file provides the structure for your project, linking together the p5.js library, and another file called *sketch.js*, which is where you will write your own program. The code that creates these links look like this:

```
<script language="javascript" type="text/javascript" src="../p5.js"></script>  
<script language="javascript" type="text/javascript" src="sketch.js"></script>
```

You don't need to do anything with the HTML file at this point—it's all set up for you. Next, click *sketch.js* and take a look at the code:

```
function setup() {  
  // put setup code here  
}  
  
function draw() {  
  // put drawing code here  
}
```

The template code contains two blocks, or functions, `setup()` and `draw()`. You can put code in either place, and there is a specific purpose for each.

Any code involved in setting up the initial state of your program goes in the `setup()` block. For now, we'll leave it empty, but later in the book, you'll add code here to set the size of your graphics canvas, the weight of your stroke, or the speed of your program.

Any code involved in actually drawing to the screen (setting the background color, or drawing shapes, text, or images) will be placed in the `draw()` block. This is where you'll begin writing your first lines of code.

Example 2-1: Draw an Ellipse

Within the curly braces of the `draw()` block, delete the text *// put drawing code here* and replace it with the following:

```
background(204);  
ellipse(50, 50, 80, 80);
```

Your full program should look like this:

```
function setup() {  
  // put setup code here  
}  
  
function draw() {  
  background(204);  
  ellipse(50, 50, 80, 80);  
}
```

This new line of code means “draw an ellipse, with the center 50 pixels over from the left and 50 pixels down from the top, with a width and height of 80 pixels.” Save the code by pressing Command-S, or choosing File→Save from the menu.

To view the running code, you can open the *index.html* file in any web browser (like Chrome, Firefox, or Safari). Navigate to the *empty-example* folder in your filesystem, and double-click *index.html* to open it. Alternatively, in your browser, choose File→Open and select the *index.html* file.

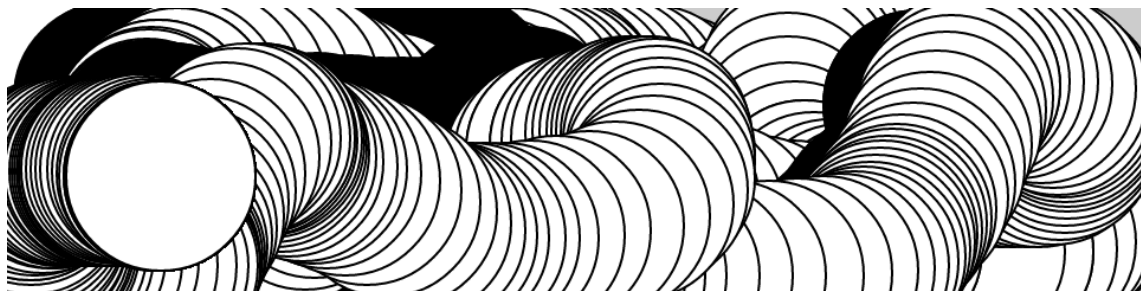
If you've typed everything correctly, you'll see a circle in the browser. If you don't see it, make sure that you've copied the example code exactly. The numbers should be contained within parentheses and have commas between each of them. The line should end with a semicolon.

One of the most difficult things about getting started with programming is that you have to be very specific about the syntax. The p5.js software isn't always smart enough to know what you mean, and can be quite fussy about the placement of punctuation. You'll get used to it with a little practice.

Next, we'll skip ahead to a sketch that's a little more exciting.

Example 2-2: Make Circles

Delete the text from the last example, and try this one. Save your code, and reopen or refresh (Command-R) *index.html* in your browser to see it update.



```
function setup() {  
  createCanvas(480, 120);  
}  
  
function draw() {  
  if (mouseIsPressed) {  
    fill(0);  
  } else {  
    fill(255);  
  }  
  ellipse(mouseX, mouseY, 80, 80);  
}
```

This program creates a graphics canvas that is 480 pixels wide and 120 pixels high, and then starts drawing white circles at the position of the mouse. When a mouse button is pressed, the circle color changes to black. We'll explain more about the elements of this program in detail later. For now, run the code, move the mouse, and click to experience it.

The Console

The browser comes with a built-in *console* that can be very useful for debugging programs. Each browser has a different way to open the console. Here's how to do it in some of the most common browsers:

- To open the console with Chrome, from the top menu select View→Developer→JavaScript Console.
- With Firefox, from the top menu select Tools→Web Developer→Web Console.
- Using Safari, you'll need to enable the functionality before you can use it. From the top menu, select Preferences, then click the Advanced tab and check the box next to the text "Show Develop menu in menu bar." Once you've done this, you'll be able to select Develop→Show Error Console.
- In Internet Explorer, open the F12 Developer Tools, then select the Console tool.

You should now see a box appear at the bottom or side of your screen ([Figure 2-1](#)). If there is a typo or other error in your program, you may see some red text explaining what the error is. This text can sometimes be a bit cryptic, but if you look to the righthand side of the line, you will notice a filename and line number where the error is detected. This is a good place to look first for errors in your program.

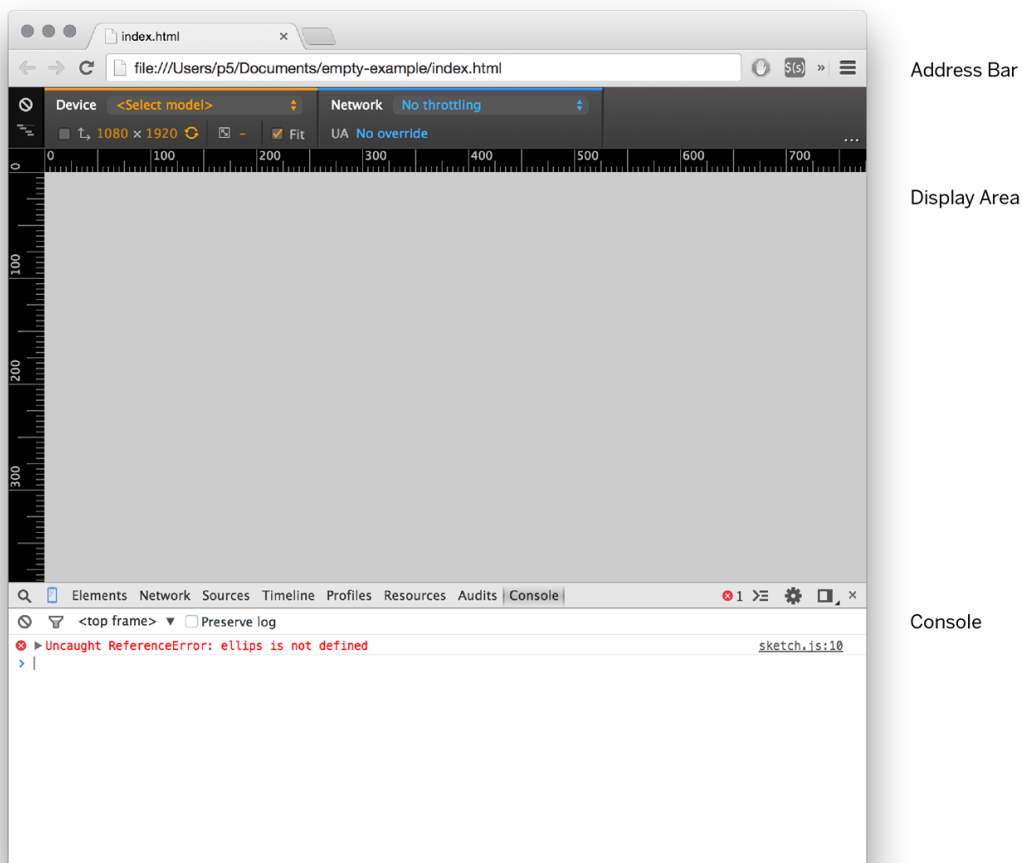


Figure 2-1. Example view of an error in the console (the appearance and layout will vary based on the browser used)

Making a New Project

You've created one sketch from the empty example, but how do you make a new project? The easiest way to do this is to locate the *empty-example* folder in your filesystem, then copy and paste it to create a second *empty-example*. You can rename the folder to anything you like—for example, *Project-2*.

You can now open this folder in your code editor and begin making a new sketch. When you want to view it in the browser, open the *index.html* within your new *Project-2* folder.

It's always a good idea to save your sketches often. As you try different things, keep saving with different names (File→Save As), so that you can always go back to an earlier version. This is especially helpful if—no, *when*—something breaks.

NOTE

A common mistake is to be editing one project but viewing a different one in the browser, preventing any of your changes from showing up. If you notice that your program looks the same despite changes to your code, double-check that you are viewing the right *index.html* file.

Examples and Reference

Learning how to program with p5.js involves exploring lots of code: running, altering, breaking, and enhancing it until you have reshaped it into something new. With this in mind, the p5.js website has dozens of examples that demonstrate different features of the library. Visit [the Examples page](#) to see them. You can play with them by editing the code of each example on the page and clicking “run.” The examples are grouped into categories based on their function, such as Form, Color, and Image. Find an interesting topic in the list and try an example.

If you see a part of the program you're unfamiliar with or want to learn more about its functionality, visit [the p5.js Reference](#).

The *p5.js Reference* explains every code element with a description and examples. The *Reference* programs are much shorter (usually four or five lines) and easier to follow than the examples on the Learn page. Note that these examples often omit `setup()` and `draw()` for simplicity, but the lines you see there are intended to be put inside one of these blocks in order to run. We recommend keeping the *Reference* open while you're reading this book and while you're programming. It can be navigated by topic or by using the search bar at the top of the page.

The *Reference* was written with the beginner in mind; we hope that we've made it clear and understandable. We're grateful to the many people who've spotted errors and reported them. If you think you can improve a reference entry or that you've found a mistake, please let us know by clicking the link at the bottom of each reference page.

Chapter 3. Draw

At first, drawing on a computer screen is like working on graph paper. It starts as a careful technical procedure, but as new concepts are introduced, drawing simple shapes with software expands into animation and interaction. Before we make this jump, we need to start at the beginning.

A computer screen is a grid of light elements called *pixels*. Each pixel has a position within the grid defined by coordinates. When you create a p5.js sketch, you view it with a web browser. Within the window of the browser, p5.js creates a *drawing canvas*, an area in which graphics are drawn. The canvas may be the same size as the window, or it may have different dimensions. The canvas is usually positioned at the top left of your window, but you can position it in other locations.

When drawing on the canvas, the *x* coordinate is the distance from the left edge of the canvas and the *y* coordinate is the distance from the top edge. We write coordinates of a pixel like this: (*x*, *y*). So, if the canvas is 200×200 pixels, the upper left is (0, 0), the center is at (100, 100), and the lower right is (199, 199). These numbers may seem confusing; why do we go from 0 to 199 instead of 1 to 200? The answer is that in code, we usually count from 0 because it's easier for calculations that we'll get into later.

The Canvas

The canvas is created and images are drawn inside through code elements called *functions*. Functions are the basic building blocks of a p5.js program. The behavior of a function is defined by its *parameters*. For example, almost every p5.js program has a `createCanvas()` function that creates a drawing canvas with a specific width and height. If your program doesn't have a `createCanvas()` function, a canvas with dimensions 100×100 pixels is created.

Example 3-1: Create a Canvas

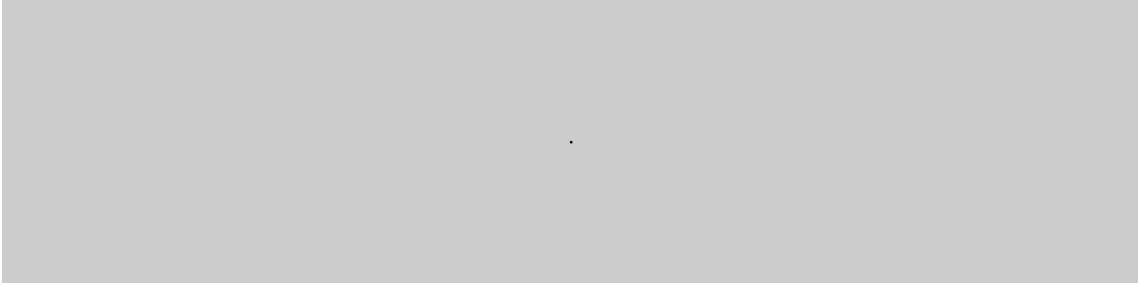
The `createCanvas()` function has two parameters; the first sets the width of the drawing canvas, and the second sets the height. To draw a canvas that is 800 pixels wide and 600 pixels high, type:

```
function setup() {  
  createCanvas(800, 600);  
}
```

Run this line of code to see the result. Put in different values to see what's possible. Try very small numbers and numbers larger than your screen.

Example 3-2: Draw a Point

To set the color of a single pixel within the canvas, we use the `point()` function. It has two parameters that define a position: the *x* coordinate followed by the *y* coordinate. To create a small canvas and a point at the center of it, coordinate (240, 60), type:



```
function setup() {  
  createCanvas(480, 120);  
}
```

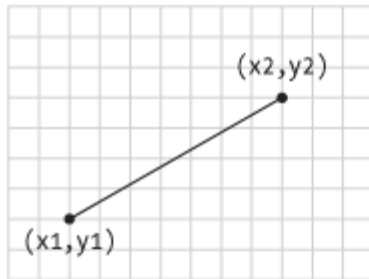
```
function draw() {  
  background(204);  
  point(240, 60);  
}
```

Try to write a program that puts a point at each corner of the drawing canvas and one in the center. Then take a stab at placing points side by side to make horizontal, vertical, and diagonal lines.

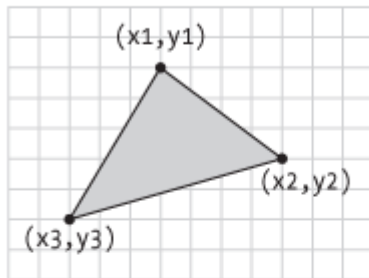
Basic Shapes

p5.js includes a group of functions to draw basic shapes (see [Figure 3-1](#)). Simple shapes like lines can be combined to create more complex forms like a leaf or a face.

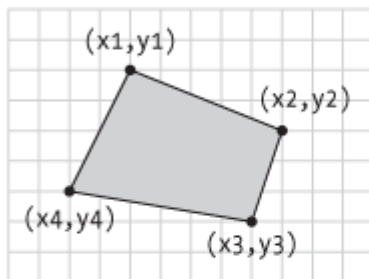
To draw a single line, we need four parameters: two for the starting location and two for the end.



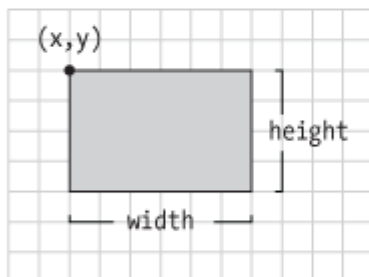
`line(x1, y1, x2, y2)`



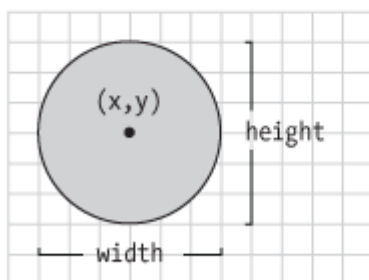
`triangle(x1, y1, x2, y2, x3, y3)`



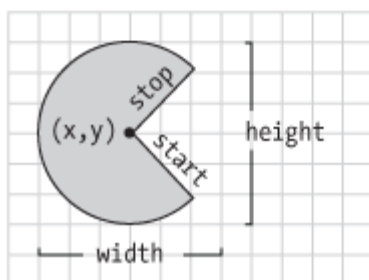
`quad(x1, y1, x2, y2, x3, y3, x4, y4)`



`rect(x, y, width, height)`



`ellipse(x, y, width, height)`

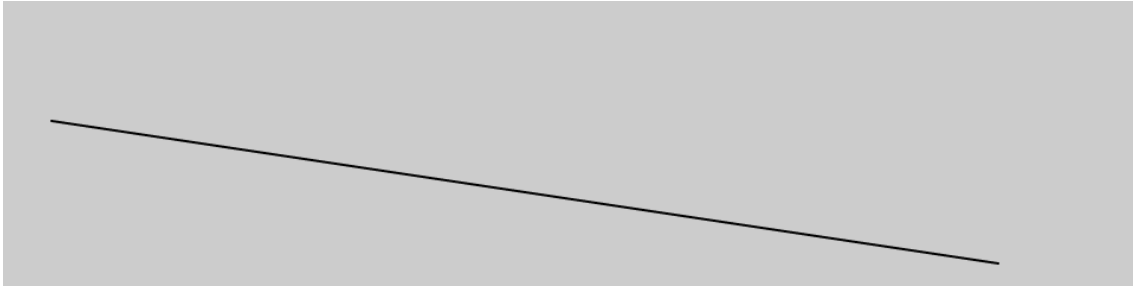


`arc(x, y, width, height, start, stop)`

Figure 3-1. Shapes and their coordinates

Example 3-3: Draw a Line

To draw a line between coordinate (20, 50) and (420, 110), try:

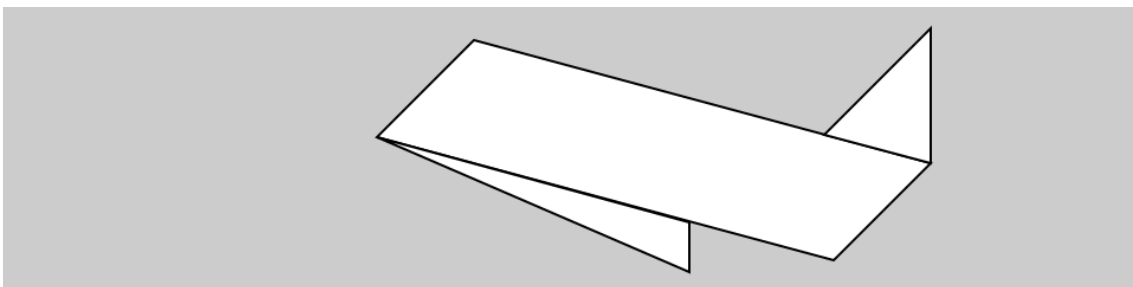


```
function setup() {  
  createCanvas(480, 120);  
}
```

```
function draw() {  
  background(204);  
  line(20, 50, 420, 110);  
}
```

Example 3-4: Draw Basic Shapes

Following this pattern, a triangle needs six parameters and a quadrilateral needs eight (one pair for each point):

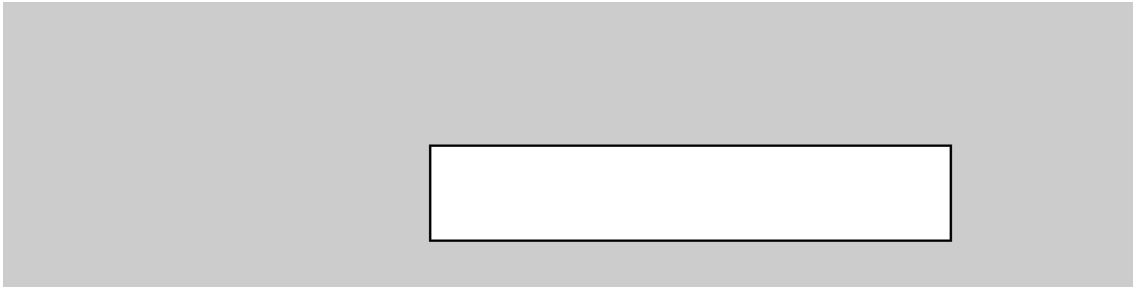


```
function setup() {  
  createCanvas(480, 120);  
}
```

```
function draw() {  
  background(204);  
  quad(158, 55, 199, 14, 392, 66, 351, 107);  
  triangle(347, 54, 392, 9, 392, 66);  
  triangle(158, 55, 290, 91, 290, 112);  
}
```

Example 3-5: Draw a Rectangle

Rectangles and *ellipses* are both defined with four parameters: the first and second are the x and y coordinates of the anchor point, the third for the width, and the fourth for the height. To make a rectangle at coordinate (180, 60) with a width of 220 pixels and height of 40, use the `rect()` function like this:

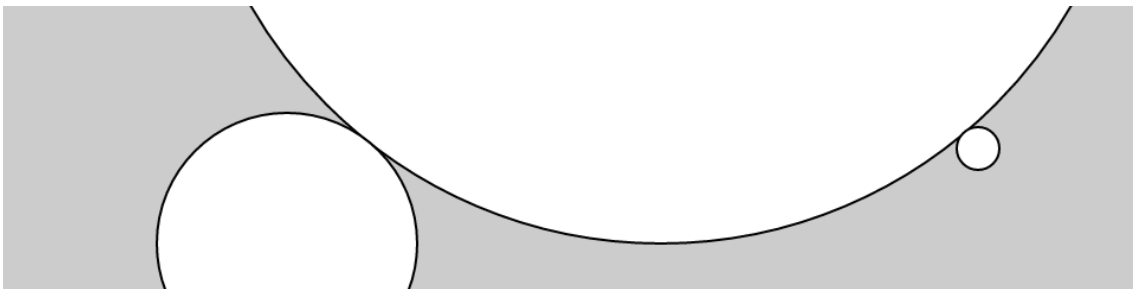


```
function setup() {  
  createCanvas(480, 120);  
}
```

```
function draw() {  
  background(204);  
  rect(180, 60, 220, 40);  
}
```

Example 3-6: Draw an Ellipse

The x and y coordinates for a rectangle are the upper-left corner, but for an ellipse they are the center of the shape. In this example, notice that the y coordinate for the first ellipse is outside the canvas. Objects can be drawn partially (or entirely) out of the canvas without an error:



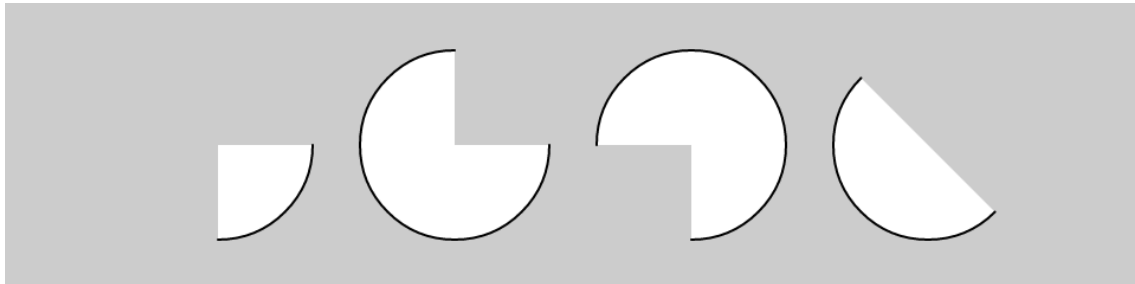
```
function setup() {  
  createCanvas(480, 120);  
}
```

```
function draw() {  
  background(204);  
  ellipse(278, -100, 400, 400);  
  ellipse(120, 100, 110, 110);  
  ellipse(412, 60, 18, 18);  
}
```

p5.js doesn't have separate functions to make squares and circles. To make these shapes, use the same value for the *width* and the *height* parameters for `ellipse()` and `rect()`.

Example 3-7: Draw Part of an Ellipse

The `arc()` function draws a piece of an ellipse:



```
function setup() {  
  createCanvas(480, 120);  
}  
  
function draw() {  
  background(204);  
  arc(90, 60, 80, 80, 0, HALF_PI);  
  arc(190, 60, 80, 80, 0, PI+HALF_PI);  
  arc(290, 60, 80, 80, PI, TWO_PI+HALF_PI);  
  arc(390, 60, 80, 80, QUARTER_PI, PI+QUARTER_PI);  
}
```

The first and second parameters set the location, while the third and fourth set the width and height. The fifth parameter sets the angle to start the arc and the sixth sets the angle to stop. The angles are set in radians, rather than degrees. *Radians* are angle measurements based on the value of pi (3.14159). [Figure 3-2](#) shows how the two relate. As featured in this example, four radian values are used so frequently that special names for them were added as a part of p5.js. The values `PI`, `QUARTER_PI`, `HALF_PI`, and `TWO_PI` can be used to replace the radian values for 180°, 45°, 90°, and 360°.

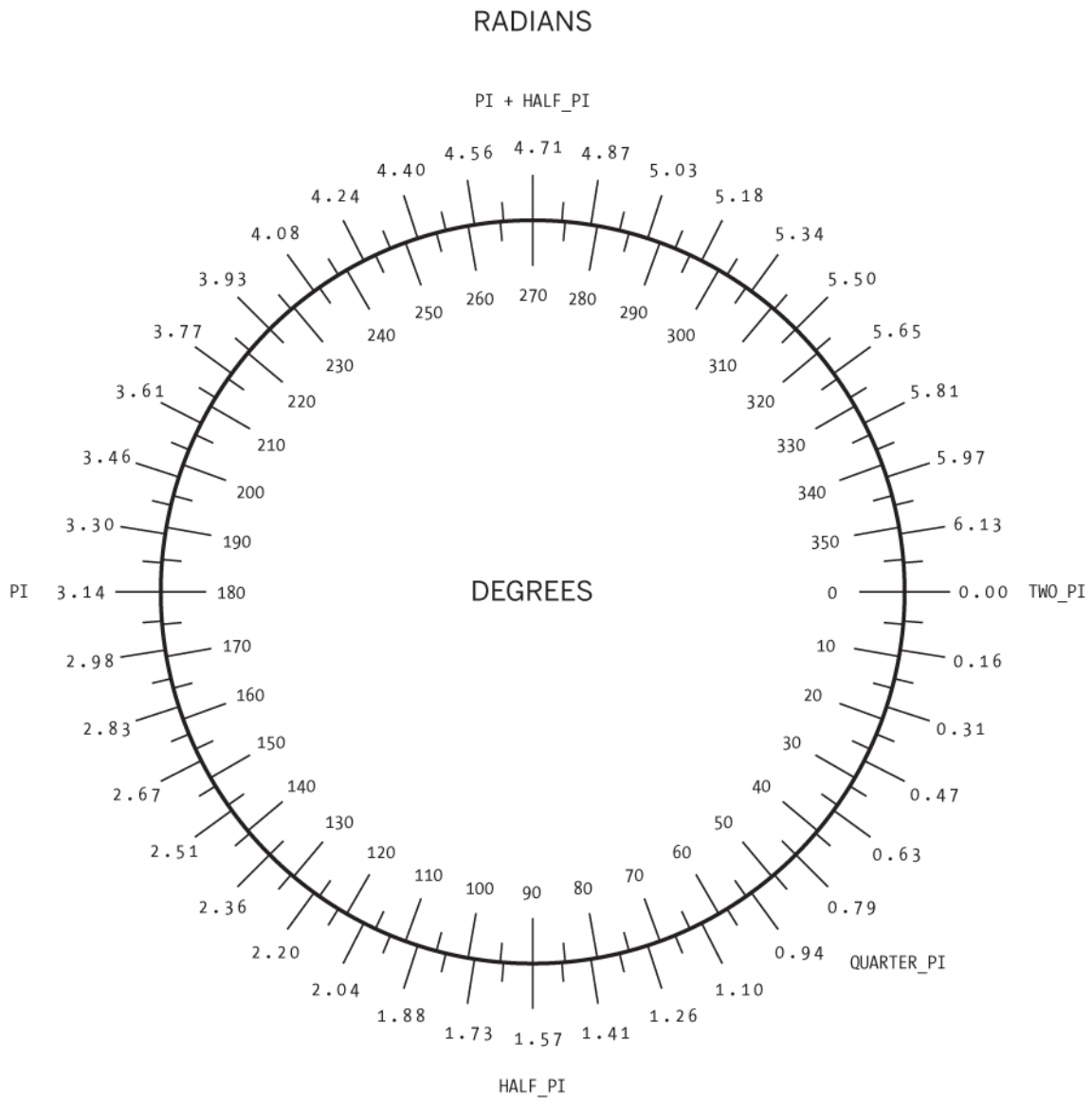


Figure 3-2. Radians and degrees are two ways to measure an angle. Degrees move around the circle from 0 to 360, while radians measure the angles in relation to pi, from 0 to approximately 6.28.

Example 3-8: Draw with Degrees

If you prefer to use degree measurements, you can convert to radians with the `radians()` function. This function takes an angle in degrees and changes it to the corresponding radian value. The following example is the same as [Example 3-7](#), but it uses the `radians()` function to define the start and stop values in degrees:

```
function setup() {
  createCanvas(480, 120);
}
```

```
function draw() {
  background(204);
```

```
arc(90, 60, 80, 80, 0, radians(90));
arc(190, 60, 80, 80, 0, radians(270));
arc(290, 60, 80, 80, radians(180), radians(450));
arc(390, 60, 80, 80, radians(45), radians(225));
}
```

Example 3-9: Use angleMode

Alternatively, you can convert your entire sketch to use degrees instead of radians using the `angleMode()` function. This changes all functions that accept or return angles to use degrees or radians based on which parameter is passed in, instead of you needing to convert them. The following example is the same as [Example 3-8](#), but it uses the `angleMode(DEGREES)` function to define the start and stop values in degrees:

```
function setup() {
  createCanvas(480, 120);
  angleMode(DEGREES);
}

function draw() {
  background(204);
  arc(90, 60, 80, 80, 0, 90);
  arc(190, 60, 80, 80, 0, 270);
  arc(290, 60, 80, 80, 180, 450);
  arc(390, 60, 80, 80, 45, 225);
}
```

Drawing Order

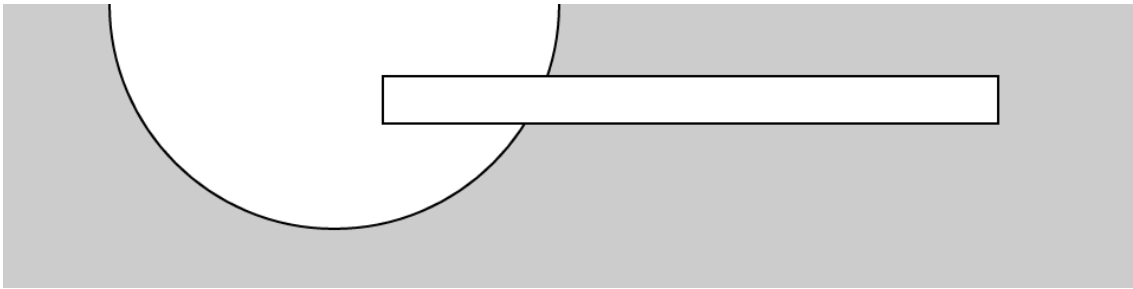
When a program runs, the computer starts at the top and reads each line of code until it reaches the last line and then stops.

NOTE

There are a few exceptions to this when it comes to loading external files, which we will get into later. For now, you can assume each line runs in order when drawing.

If you want a shape to be drawn on top of all other shapes, it needs to follow the others in the code.

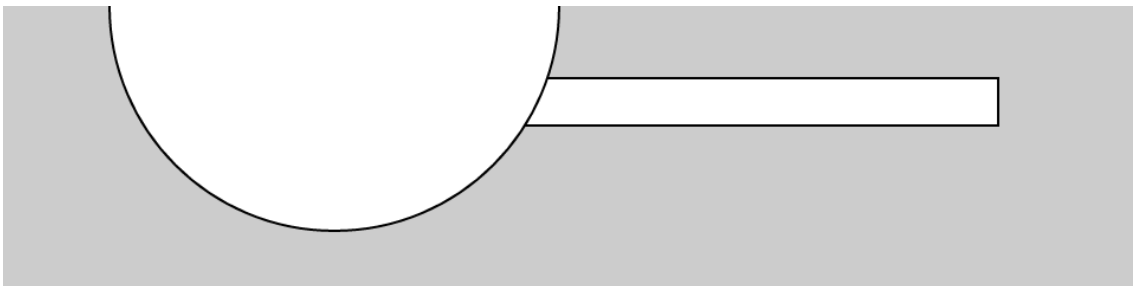
Example 3-10: Control Your Drawing Order



```
function setup() {  
  createCanvas(480, 120);  
}  
  
function draw() {  
  background(204);  
  ellipse(140, 0, 190, 190);  
  // The rectangle draws on top of the ellipse  
  // because it comes after in the code  
  rect(160, 30, 260, 20);  
}
```

Example 3-11: Put It in Reverse

Modify by reversing the order of `rect()` and `ellipse()` to see the circle on top of the rectangle:



```
function setup() {  
  createCanvas(480, 120);  
}  
  
function draw() {  
  background(204);  
  rect(160, 30, 260, 20);  
  // The ellipse draws on top of the rectangle  
  // because it comes after in the code  
  ellipse(140, 0, 190, 190);  
}
```

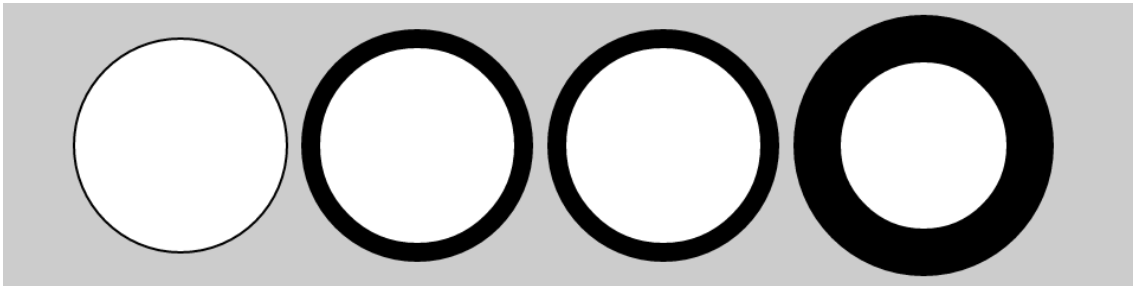
You can think of it like painting with a brush or making a collage. The last element that you add is what's visible on top.

Shape Properties

You may want to have further control over the shapes you draw, beyond just position and size. To do this, there is a set of functions to set shape properties.

Example 3-12: Set Stroke Weight

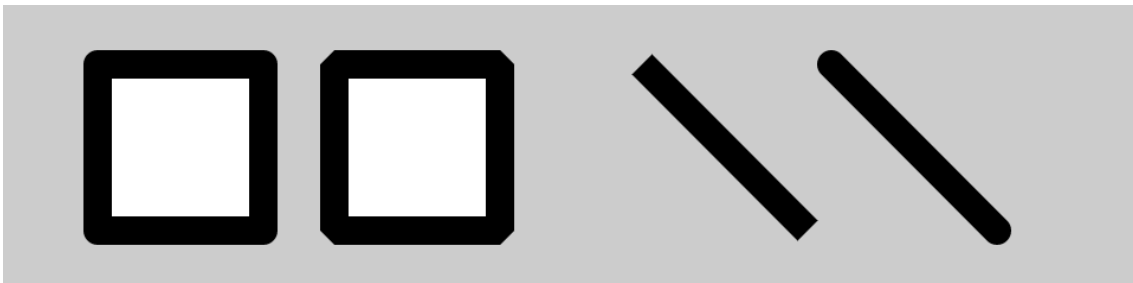
The default stroke weight is a single pixel, but this can be changed with the `strokeWeight()` function. The single parameter to `strokeWeight()` sets the width of drawn lines:



```
function setup() {  
  createCanvas(480, 120);  
}  
  
function draw() {  
  background(204);  
  ellipse(75, 60, 90, 90);  
  strokeWeight(8); // Stroke weight to 8 pixels  
  ellipse(175, 60, 90, 90);  
  ellipse(279, 60, 90, 90);  
  strokeWeight(20); // Stroke weight to 20 pixels  
  ellipse(389, 60, 90, 90);  
}
```

Example 3-13: Set Stroke Attributes

The `strokeJoin()` function changes the way lines are joined (how the corners look), and the `strokeCap()` function changes how lines are drawn at their beginning and end:



```
function setup() {  
  createCanvas(480, 120);  
  strokeWeight(12);  
}
```

```
function draw() {  
  background(204);  
  strokeJoin(ROUND); // Round the stroke corners  
  rect(40, 25, 70, 70);  
  strokeJoin(BEVEL); // Bevel the stroke corners  
  rect(140, 25, 70, 70);  
  strokeCap(SQUARE); // Square the line endings  
  line(270, 25, 340, 95);  
  strokeCap(ROUND); // Round the line endings  
  line(350, 25, 420, 95);  
}
```

The placement of shapes like `rect()` and `ellipse()` are controlled with the `rectMode()` and `ellipseMode()` functions. Check the *p5.js Reference* to see examples of how to place rectangles from their center (rather than their upper-left corner), or to draw ellipses from their upper-left corner like rectangles.

When any of these attributes are set, all shapes drawn afterward are affected. For instance, in [Example 3-12](#), notice how the second and third circles both have the same stroke weight, even though the weight is set only once before both are drawn.

Notice that the `strokeWeight(12)` line appears in `setup()` instead of in `draw()`. This is because it doesn't change at all in our program, so we can just set it once and for all in `setup()`. This is more for organization; placing the line in `draw()` would have the same visual effect.

Color

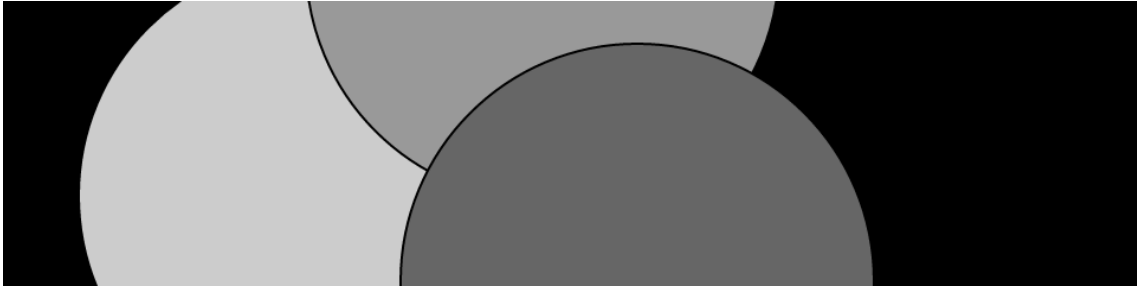
All the shapes so far have been filled white with black outlines. To change this, use the `fill()` and `stroke()` functions. The values of the parameters range from 0 to 255, where 255 is white, 128 is medium gray, and 0 is black. [Figure 3-3](#) shows how the values from 0 to 255 map to different gray levels. The `background()` function we've seen in previous examples works in the same way, except rather than setting the fill or stroke color for drawing, it sets the background color of the canvas.

0	64	128	192
1	65	129	193
2	66	130	194
3	67	131	195
4	68	132	196
5	69	133	197
6	70	134	198
7	71	135	199
8	72	136	200
9	73	137	201
10	74	138	202
11	75	139	203
12	76	140	204
13	77	141	205
14	78	142	206
15	79	143	207
16	80	144	208
17	81	145	209
18	82	146	210
19	83	147	211
20	84	148	212
21	85	149	213
22	86	150	214
23	87	151	215
24	88	152	216
25	89	153	217
26	90	154	218
27	91	155	219
28	92	156	220
29	93	157	221
30	94	158	222
31	95	159	223
32	96	160	224
33	97	161	225
34	98	162	226
35	99	163	227
36	100	164	228
37	101	165	229
38	102	166	230
39	103	167	231
40	104	168	232
41	105	169	233
42	106	170	234
43	107	171	235
44	108	172	236
45	109	173	237
46	110	174	238
47	111	175	239
48	112	176	240
49	113	177	241
50	114	178	242
51	115	179	243
52	116	180	244
53	117	181	245
54	118	182	246
55	119	183	247
56	120	184	248
57	121	185	249
58	122	186	250
59	123	187	251
60	124	188	252
61	125	189	253
62	126	190	254
63	127	191	255

Figure 3-3. Gray values from 0 to 255

Example 3-14: Paint with Grays

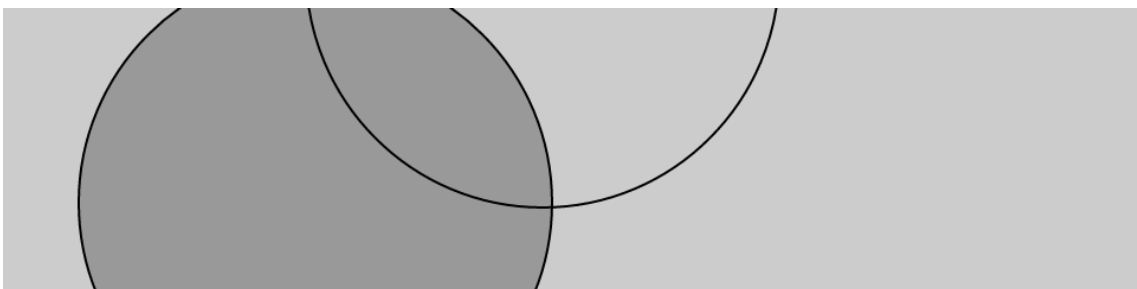
This example shows three different gray values on a black background:



```
function setup() {  
  createCanvas(480, 120);  
}  
  
function draw() {  
  background(0); // Black  
  fill(204); // Light gray  
  ellipse(132, 82, 200, 200); // Light gray circle  
  fill(153); // Medium gray  
  ellipse(228, -16, 200, 200); // Medium gray circle  
  fill(102); // Dark gray  
  ellipse(268, 118, 200, 200); // Dark gray circle  
}
```

Example 3-15: Control Fill and Stroke

You can use `noStroke()` to disable the stroke so that there's no outline, and you can disable the fill of a shape with `noFill()`:



```
function setup() {  
  createCanvas(480, 120);  
}  
  
function draw() {  
  background(204);  
  fill(153); // Medium gray  
  ellipse(132, 82, 200, 200); // Gray circle  
}
```

```

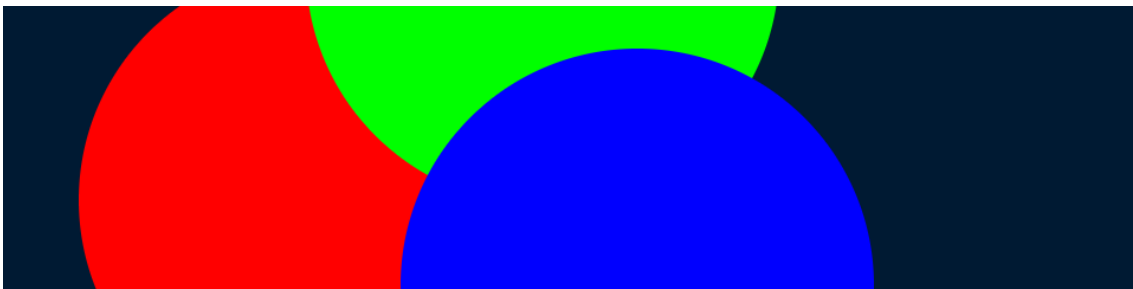
noFill();           // Turn off fill
ellipse(228, -16, 200, 200); // Outline circle
noStroke();        // Turn off stroke
ellipse(268, 118, 200, 200); // Doesn't draw!
}

```

Be careful not to disable the fill and stroke at the same time, as we've done in the previous example, because nothing will draw to the screen.

Example 3-16: Draw with Color

To move beyond grayscale values, you use three parameters to specify the red, green, and blue components of a color. Because this book is printed in black and white, you'll see only gray values here. Run the code to reveal the colors:



```

function setup() {
  createCanvas(480, 120);
  noStroke();
}

function draw() {
  background(0, 26, 51); // Dark blue color
  fill(255, 0, 0);      // Red color
  ellipse(132, 82, 200, 200); // Red circle
  fill(0, 255, 0);     // Green color
  ellipse(228, -16, 200, 200); // Green circle
  fill(0, 0, 255);     // Blue color
  ellipse(268, 118, 200, 200); // Blue circle
}

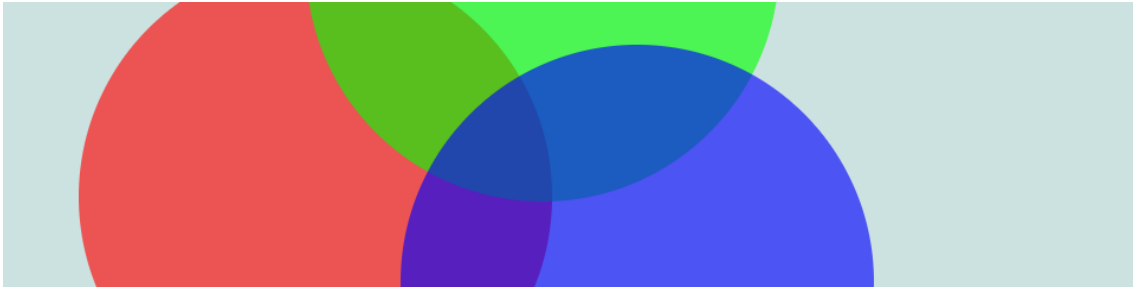
```

The colors in the example are referred to as *RGB color*, which is how computers define colors on the screen. The three numbers stand for the values of red, green, and blue, and they range from 0 to 255 the way that the gray values do. These three numbers are the parameters for your `background()`, `fill()`, and `stroke()` functions.

Example 3-17: Set Transparency

By adding an optional fourth parameter to `fill()` or `stroke()`, you can control the transparency. This fourth parameter is known as the alpha value, and also uses the range 0 to 255 to set the amount of transparency. The value 0 defines the color as entirely

transparent (it won't display), the value 255 is entirely opaque, and the values between these extremes cause the colors to mix on screen:



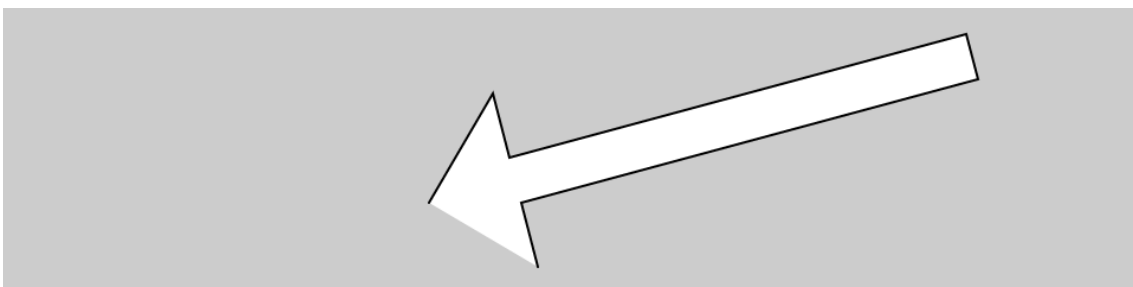
```
function setup() {  
  createCanvas(480, 120);  
  noStroke();  
}  
  
function draw() {  
  background(204, 226, 225); // Light blue color  
  fill(255, 0, 0, 160); // Red color  
  ellipse(132, 82, 200, 200); // Red circle  
  fill(0, 255, 0, 160); // Green color  
  ellipse(228, -16, 200, 200); // Green circle  
  fill(0, 0, 255, 160); // Blue color  
  ellipse(268, 118, 200, 200); // Blue circle  
}
```

Custom Shapes

You're not limited to using these basic geometric shapes—you can also define new shapes by connecting a series of points.

Example 3-18: Draw an Arrow

The `beginShape()` function signals the start of a new shape. The `vertex()` function is used to define each pair of x and y coordinates for the shape. Finally, `endShape()` is called to signal that the shape is finished:

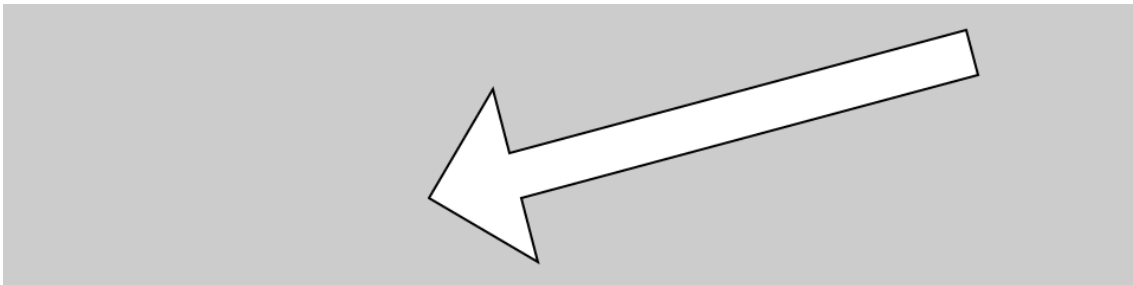


```
function setup() {  
  createCanvas(480, 120);  
}
```

```
function draw() {  
  background(204);  
  beginShape();  
  vertex(180, 82);  
  vertex(207, 36);  
  vertex(214, 63);  
  vertex(407, 11);  
  vertex(412, 30);  
  vertex(219, 82);  
  vertex(226, 109);  
  endShape();  
}
```

Example 3-19: Close the Gap

When you run [Example 3-18](#), you'll see the first and last point are not connected. To do this, add the word CLOSE as a parameter to endShape(), like this:



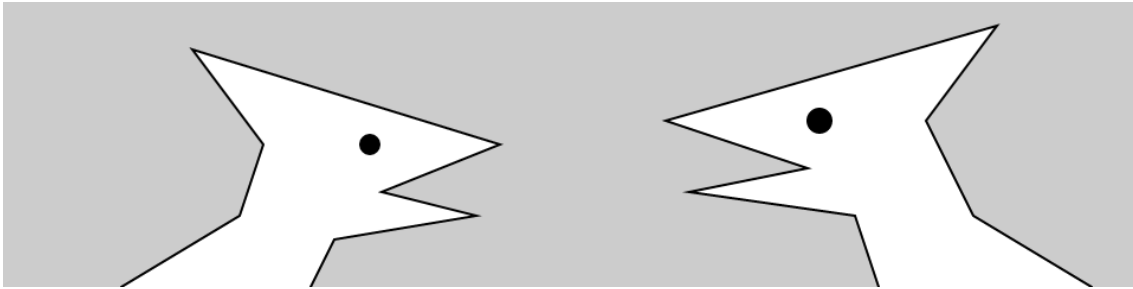
```
function setup() {  
  createCanvas(480, 120);  
}
```

```
function draw() {  
  background(204);  
  beginShape();  
  vertex(180, 82);  
  vertex(207, 36);  
  vertex(214, 63);  
  vertex(407, 11);  
  vertex(412, 30);  
  vertex(219, 82);  
  vertex(226, 109);  
  endShape(CLOSE);  
}
```

Example 3-20: Create Some Creatures

The power of defining shapes with vertex() is the ability to make shapes with complex outlines. p5.js can draw thousands and thousands of lines at a time to fill the screen with

fantastic shapes that spring from your imagination. A modest but more complex example follows:



```
function setup() {  
  createCanvas(480, 120);  
}
```

```
function draw() {  
  background(204);  
  
  // Left creature  
  beginShape();  
  vertex(50, 120);  
  vertex(100, 90);  
  vertex(110, 60);  
  vertex(80, 20);  
  vertex(210, 60);  
  vertex(160, 80);  
  vertex(200, 90);  
  vertex(140, 100);  
  vertex(130, 120);  
  endShape();  
  fill(0);  
  ellipse(155, 60, 8, 8);  
  
  // Right creature  
  fill(255);  
  beginShape();  
  vertex(370, 120);  
  vertex(360, 90);  
  vertex(290, 80);  
  vertex(340, 70);  
  vertex(280, 50);  
  vertex(420, 10);  
  vertex(390, 50);  
  vertex(410, 90);  
  vertex(460, 120);  
  endShape();  
  fill(0);  
  ellipse(345, 50, 10, 10);  
}
```

Comments

The examples in this chapter use double slashes (//) at the end of a line to add comments to the code. *Comments* are parts of the program that are ignored when the program is run. They are useful for making notes for yourself that explain what's happening in the code. If others are reading your code, comments are especially important to help them understand your thought process.

Comments are also especially useful for a number of different options, such as trying to choose the right color. So, for instance, I might be trying to find just the right red for an ellipse:

```
function setup() {  
  createCanvas(200, 200);  
}
```

```
function draw() {  
  background(204);  
  fill(165, 57, 57);  
  ellipse(100, 100, 80, 80);  
}
```

Now suppose I want to try a different red, but don't want to lose the old one. I can copy and paste the line, make a change, and then "comment out" the old one:

```
function setup() {  
  createCanvas(200, 200);  
}
```

```
function draw() {  
  background(204);  
  //fill(165, 57, 57);  
  fill(144, 39, 39);  
  ellipse(100, 100, 80, 80);  
}
```

Placing // at the beginning of the line temporarily disables it. Or I can remove the // and place it in front of the other line if I want to try it again:

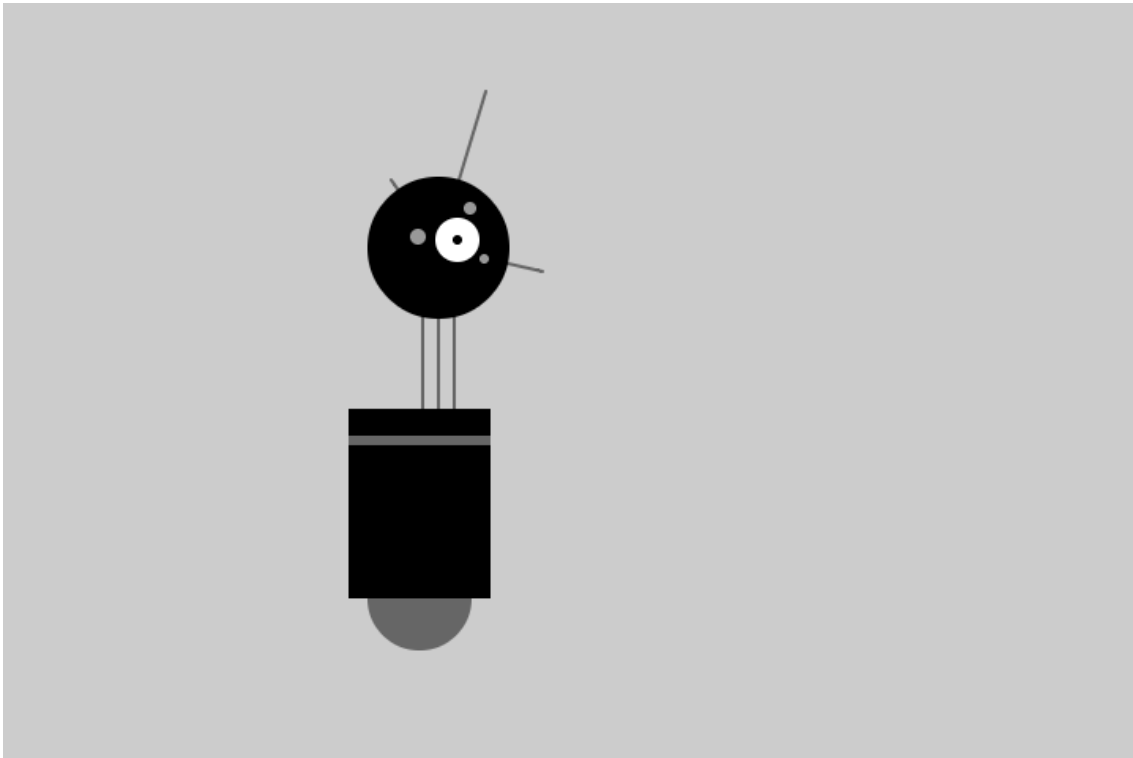
```
function setup() {  
  createCanvas(200, 200);  
}
```

```
function draw() {  
  background(204);  
  fill(165, 57, 57);  
  //fill(144, 39, 39);  
}
```

```
ellipse(100, 100, 80, 80);  
}
```

As you work with p5.js sketches, you'll find yourself creating dozens of iterations of ideas; using comments to make notes or to disable code can help you keep track of multiple options.

Robot 1: Draw



This is P5, the p5.js Robot. There are 10 different programs to draw and animate her in this book—each one explores a different programming idea. P5's design was inspired by Sputnik I (1957), Shakey from the Stanford Research Institute (1966–1972), the fighter drone in David Lynch's *Dune* (1984), and HAL 9000 from *2001: A Space Odyssey* (1968), among other robot favorites.

The first robot program uses the drawing functions introduced earlier in this chapter. The parameters to the `fill()` and `stroke()` functions set the gray values. The `line()`, `ellipse()`, and `rect()` functions define the shapes that create the robot's neck, antennae, body, and head. To get more familiar with the functions, run the program and change the values to redesign the robot:

```
function setup() {  
  createCanvas(720, 480);  
  strokeWeight(2);  
  ellipseMode(RADIUS);  
}
```

```
function draw() {
```



```

background(204);

// Neck
stroke(102);          // Set stroke to gray
line(266, 257, 266, 162); // Left
line(276, 257, 276, 162); // Middle
line(286, 257, 286, 162); // Right

// Antennae
line(276, 155, 246, 112); // Small
line(276, 155, 306, 56); // Tall
line(276, 155, 342, 170); // Medium

// Body
noStroke();          // Disable stroke
fill(102);           // Set fill to gray
ellipse(264, 377, 33, 33); // Antigravity orb
fill(0);              // Set fill to black
rect(219, 257, 90, 120); // Main body
fill(102);           // Set fill to gray
rect(219, 274, 90, 6); // Gray stripe

// Head
fill(0);              // Set fill to black
ellipse(276, 155, 45, 45); // Head
fill(255);           // Set fill to white
ellipse(288, 150, 14, 14); // Large eye
fill(0);              // Set fill to black
ellipse(288, 150, 3, 3); // Pupil
fill(153);           // Set fill to light gray
ellipse(263, 148, 5, 5); // Small eye 1
ellipse(296, 130, 4, 4); // Small eye 2
ellipse(305, 162, 3, 3); // Small eye 3
}

```

Capítulo 4. Variáveis

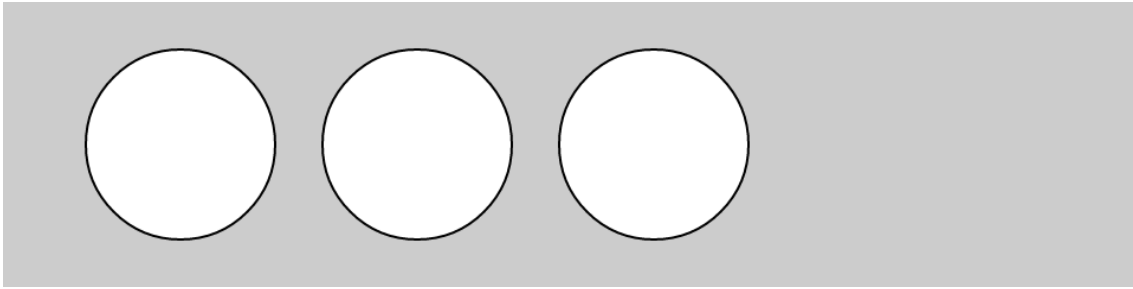
Uma *variável* armazena um valor na memória para que possa ser usado posteriormente em um programa. Uma variável pode ser usada muitas vezes em um único programa e o valor pode ser facilmente alterado enquanto o programa está em execução.

Primeiras Variáveis

A principal razão pela qual usamos variáveis é evitar nos repetirmos no código. Se você estiver digitando o mesmo número mais de uma vez, considere usar uma variável para que seu código seja mais geral e mais fácil de atualizar.

Exemplo 4-1: Reutilize os mesmos valores

ParaPor exemplo, quando você transforma a coordenada y e o diâmetro dos três círculos neste exemplo em variáveis, os mesmos valores são usados para cada elipse:



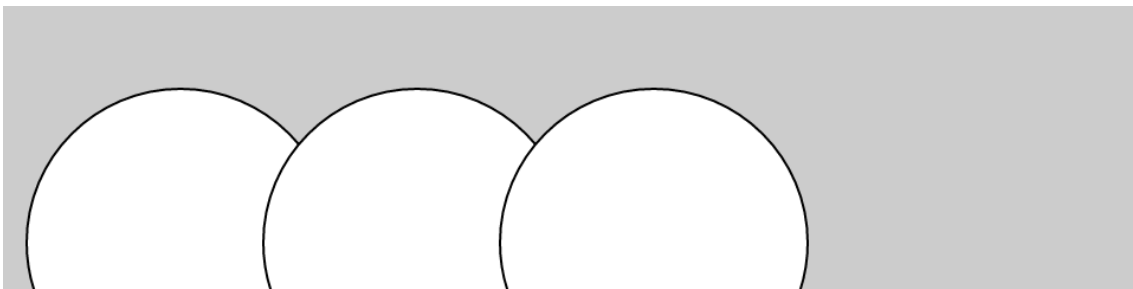
```
var y = 60;
var d = 80;

function setup() {
  createCanvas(480, 120);
}

function draw() {
  background(204);
  ellipse(75, y, d, d); // Left
  ellipse(175, y, d, d); // Middle
  ellipse(275, y, d, d); // Right
}
```

Exemplo 4-2: Alterar Valores

Simplesmente alterar as variáveis y e d altera, portanto, todas as três elipses:



```
var y = 100;
var d = 130;

function setup() {
  createCanvas(480, 120);
}

function draw() {
  background(204);
  ellipse(75, y, d, d); // Left
  ellipse(175, y, d, d); // Middle
  ellipse(275, y, d, d); // Right
}
```

```
}
```

Sem as variáveis, você precisaria alterar a coordenada *y* usada no código três vezes e o diâmetro seis vezes. Ao comparar os Exemplos [4-1](#) e [4-2](#), observe como todas as linhas são iguais, exceto que as duas primeiras linhas com as variáveis são diferentes. As variáveis permitem separar as linhas do código que mudam das linhas que não mudam, o que torna os programas mais fáceis de modificar. Por exemplo, se você colocar variáveis que controlam cores e tamanhos de formas em um só lugar, poderá explorar rapidamente diferentes opções visuais concentrando-se em apenas algumas linhas de código.

Fazendo Variáveis

Quando você cria suas próprias variáveis, determina o *nome* e o *valor*. O nome é como você decide chamar a variável. Escolha um nome que seja informativo sobre o que a variável armazena, mas seja consistente e não muito detalhado. Por exemplo, o nome da variável “raio” será mais claro do que “r” quando você olhar o código posteriormente.

As variáveis devem primeiro ser *declaradas*, que reserva espaço na memória do computador para armazenar as informações. Ao declarar uma variável, você usa *var*, para indicar que está criando uma nova variável, seguido do nome. Após o nome ser definido, um valor pode ser atribuído à variável:

```
var x; // Declare x as a variable  
x = 12; // Assign a value to x
```

Este código faz a mesma coisa, mas é mais curto:

```
var x = 12; // Declare x as a variable and assign a value
```

Os caracteres *var* são incluídos na linha de código que declara uma variável, mas não são escritos novamente. Cada vez que *var* é escrito antes do nome da variável, o computador pensa que você está tentando declarar uma nova variável. Você não pode ter duas variáveis com o mesmo nome na mesma parte do programa ([Apêndice C](#)), ou o programa pode se comportar de maneira estranha:

```
var x; // Declare x as a variable  
var x = 12; // ERROR! Can't have two variables called x here
```

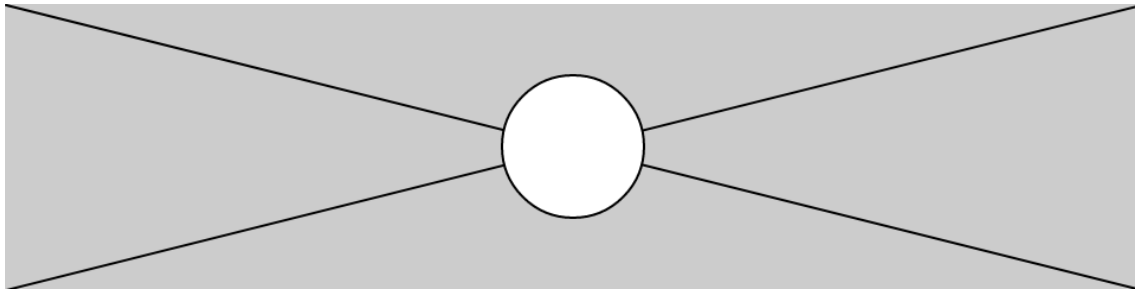
Você pode colocar suas variáveis fora de *setup()* e *draw()*. Se você criar uma variável dentro de *setup()*, não poderá usá-la dentro de *draw()*, então será necessário colocar essas variáveis em outro lugar. Tais variáveis são chamadas *variáveis globais*, porque podem ser usadas em qualquer lugar (“globalmente”) no programa.

Variáveis p5.js

p5.js tem uma série de variáveis especiais para armazenar informações sobre o programa enquanto ele é executado. Por exemplo, a largura e a altura da tela são armazenadas em variáveis chamadas `width` e `height`. Esses valores são definidos pela `createCanvas()` função. Eles podem ser usados para desenhar elementos relativos ao tamanho da tela, mesmo que a `createCanvas()` linha mude.

Exemplo 4-3: Ajuste a tela, veja o que se segue

Neste exemplo, altere os parâmetros para `createCanvas()` para ver como funciona:



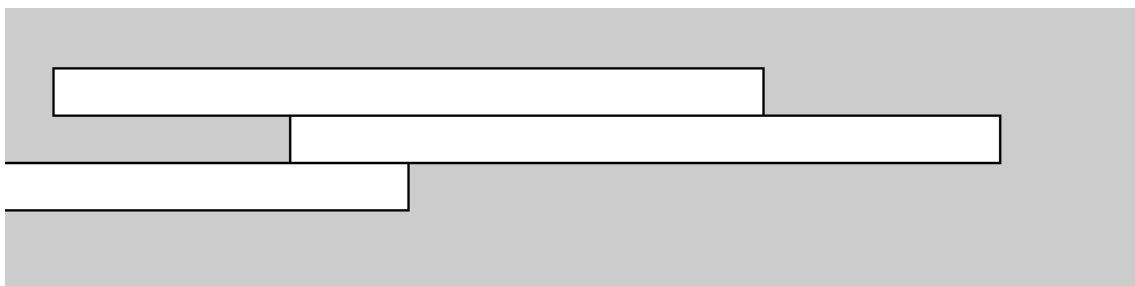
```
function setup() {  
  createCanvas(480, 120);  
}  
  
function draw() {  
  background(204);  
  line(0, 0, width, height); // Line from (0,0) to (480, 120)  
  line(width, 0, 0, height); // Line from (480, 0) to (0, 120)  
  ellipse(width/2, height/2, 60, 60);  
}
```

Outras variáveis especiais controlam o status dos valores do mouse e do teclado e muito mais. Estes são discutidos no [Capítulo 5](#).

Um pouco de matemática

Pessoas muitas vezes assumem que matemática e programação são a mesma coisa. Embora o conhecimento de matemática possa ser útil para certos tipos de codificação, a aritmética básica cobre as partes mais importantes.

Exemplo 4-4: Aritmética Básica



```
var x = 25;
```

```

var h = 20;
var y = 25;

function setup() {
  createCanvas(480, 120);
}

function draw() {
  background(204);
  x = 20;
  rect(x, y, 300, h);    // Top
  x = x + 100;
  rect(x, y + h, 300, h); // Middle
  x = x - 250;
  rect(x, y + h*2, 300, h); // Bottom
}

```

No código, símbolos como $+$, $-$ e $*$ são chamados *operadores*. Quando colocados entre dois valores, eles criam uma *expressão*. Por exemplo, $5 + 9$ e $1024 - 512$ são ambas expressões. Os operadores para as operações matemáticas básicas são:

- + Adição
- Subtração
- * Multiplicação
- / Divisão
- = Atribuição

JavaScript possui um conjunto de regras para definir quais operadores têm precedência sobre outros, ou seja, quais cálculos são feitos primeiro, segundo, terceiro e assim por diante. Essas regras definem a ordem em que o código é executado. Um pouco de conhecimento sobre isso ajuda muito a entender como funciona uma pequena linha de código como esta:

```
var x = 4 + 4 * 5; // Assign 24 to x
```

A expressão $4 * 5$ é avaliada primeiro porque a multiplicação tem a prioridade mais alta. Segundo, 4 é adicionado ao produto de $4 * 5$ para resultar em 24 . Por último, porque *operador de atribuição* (o sinal *de igual*) tem a precedência mais baixa, o valor 24 é atribuído à variável x . Isso é esclarecido entre parênteses, mas o resultado é o mesmo:

```
var x = 4 + (4 * 5); // Assign 24 to x
```

Se quiser forçar a adição a acontecer primeiro, basta mover os parênteses. Como os parênteses têm uma precedência maior que a multiplicação, a ordem é alterada e o cálculo é afetado:

```
var x = (4 + 4) * 5; // Assign 40 to x
```

Um acrônimo para esta ordem é frequentemente ensinado nas aulas de matemática: PEMDAS, que significa Parênteses, Expoentes, Multiplicação, Divisão, Adição, Subtração, onde os parênteses têm a prioridade mais alta e a subtração a mais baixa. A ordem completa das operações encontra-se no [Apêndice B](#).

Alguns cálculos são usados com tanta frequência na programação que foram desenvolvidos atalhos; é sempre bom salvar algumas teclas digitadas. Por exemplo, você pode adicionar ou subtrair uma variável com um único operador:

```
x += 10; // This is the same as x = x + 10  
y -= 15; // This is the same as y = y - 15
```

Também é comum adicionar ou subtrair 1 de uma variável, portanto também existem atalhos para isso. Os operadores ++ e -- fazem isso:

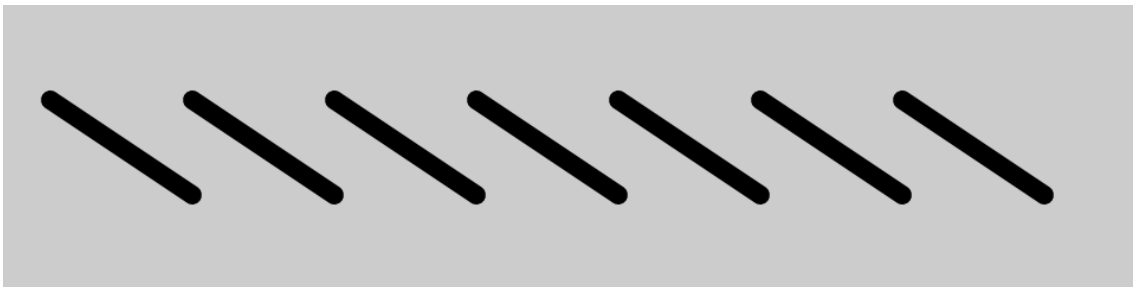
```
x++; // This is the same as x = x + 1  
y--; // This is the same as y = y - 1
```

Repetição

Como se você escrever mais programas, notará que ocorrem padrões quando as linhas de código são repetidas, mas com pequenas variações. Uma estrutura de código chamada forloop possibilita executar uma linha de código mais de uma vez para condensar esse tipo de repetição em menos linhas. Isso torna seus programas mais modulares e fáceis de alterar.

Exemplo 4-5: Faça a mesma coisa repetidamente

Este exemplo possui o tipo de padrão que pode ser simplificado com um forloop:



```
function setup() {  
  createCanvas(480, 120);
```

```

strokeWeight(8);
}

function draw() {
  background(204);
  line(20, 40, 80, 80);
  line(80, 40, 140, 80);
  line(140, 40, 200, 80);
  line(200, 40, 260, 80);
  line(260, 40, 320, 80);
  line(320, 40, 380, 80);
  line(380, 40, 440, 80);
}

```

Exemplo 4-6: Use um loop for

A mesma coisa pode ser feita com um forloop e com menos código:

```

function setup() {
  createCanvas(480, 120);
  strokeWeight(8);
}

function draw() {
  background(204);
  for (var i = 20; i < 400; i += 60) {
    line(i, 40, i + 60, 80);
  }
}

```

O forloop é diferente em muitos aspectos do código que escrevemos até agora. Observe as chaves, os caracteres { e }. O código entre chaves é chamado *bloco*. Este é o código que será repetido a cada iteração do forloop.

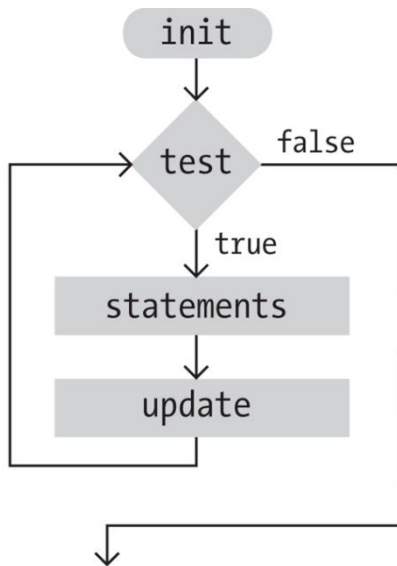
Dentro dos parênteses estão três instruções, separadas por ponto e vírgula, que funcionam juntas para controlar quantas vezes o código dentro do bloco é executado. Da esquerda para a direita, essas declarações são chamadas de a *inicialização* (*init*), o *teste* e a *atualização* :

```

for (init; test; update) {
  statements
}

```

Normalmente *init* declara uma nova variável para usar dentro do forloop e atribui um valor. O nome da variável é usado com frequência, mas não há nada de especial nisso. O *teste* avalia o valor desta variável e a *atualização* altera o valor da variável. [A Figura 4-1](#) mostra a ordem em que eles são executados e como controlam as instruções de código dentro do bloco.



```

for (init; test; update) {
    statements
}
  
```

Figura 4-1. Diagrama de fluxo de um loop for

A instrução *de teste* requer mais explicações. É sempre uma *expressão relacional* que compara dois valores com um *operador relacional*. Neste exemplo, a expressão é “i < 400” e o operador é o símbolo < (menor que). Os operadores relacionais mais comuns são:

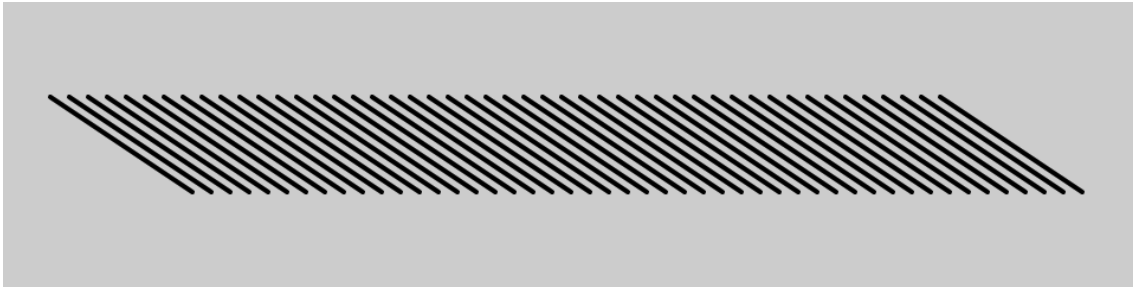
- > Maior que
- < Menor que
- >= Melhor que ou igual a
- <= Menos que ou igual a
- == Igual a
- != Não é igual a

A expressão relacional sempre é avaliada como true ou false. Por exemplo, a expressão 5 > 3 é true. Podemos fazer a pergunta: “Cinco é maior que três?” Como a resposta é “sim”, dizemos que a expressão é true. Para a expressão 5 < 3, perguntamos: “Cinco é menor que três?” Como a resposta é “não”, dizemos que a expressão é false. Quando a avaliação é true, o código dentro do bloco é executado, e quando é false, o código dentro do bloco não é executado e o for loop termina.

Exemplo 4-7: Flexione os músculos do seu loop for

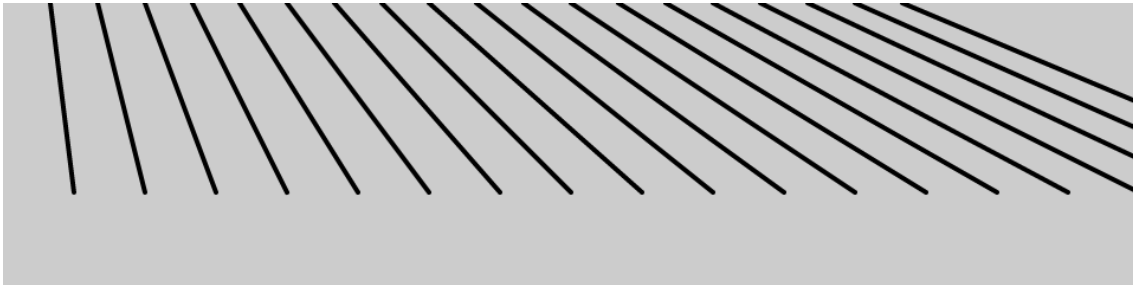
O maior poder de trabalhar com um for loop é a capacidade de fazer alterações rápidas no código. Como o código dentro do bloco normalmente é executado diversas vezes,

uma alteração no bloco é ampliada quando o código é executado. Modificando apenas ligeiramente [o Exemplo 4-6](#) , podemos criar uma série de padrões diferentes:



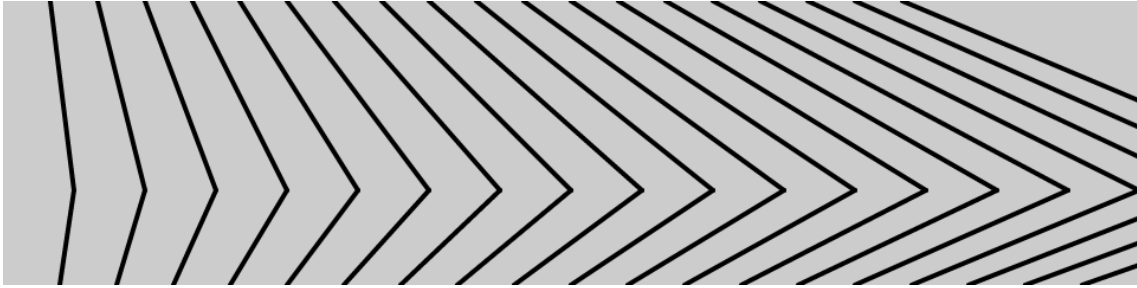
```
function setup() {  
  createCanvas(480, 120);  
  strokeWeight(2);  
}  
  
function draw() {  
  background(204);  
  for (var i = 20; i < 400; i += 8) {  
    line(i, 40, i + 60, 80);  
  }  
}
```

Exemplo 4-8: Distribuindo as Linhas



```
function setup() {  
  createCanvas(480, 120);  
  strokeWeight(2);  
}  
  
function draw() {  
  background(204);  
  for (var i = 20; i < 400; i += 20) {  
    line(i, 0, i + i/2, 80);  
  }  
}
```

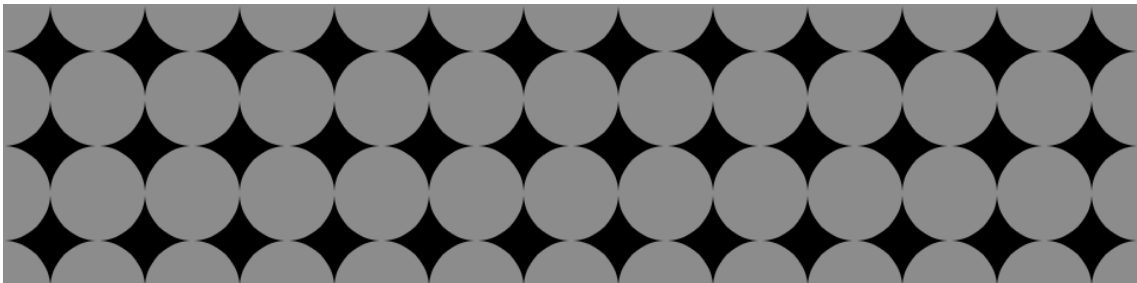
Exemplo 4-9: Dobrando as Linhas



```
function setup() {  
  createCanvas(480, 120);  
  strokeWeight(2);  
}  
  
function draw() {  
  background(204);  
  for (var i = 20; i < 400; i += 20) {  
    line(i, 0, i + i/2, 80);  
    line(i + i/2, 80, i*1.2, 120);  
  }  
}
```

Exemplo 4-10: Incorporar um loop for em outro

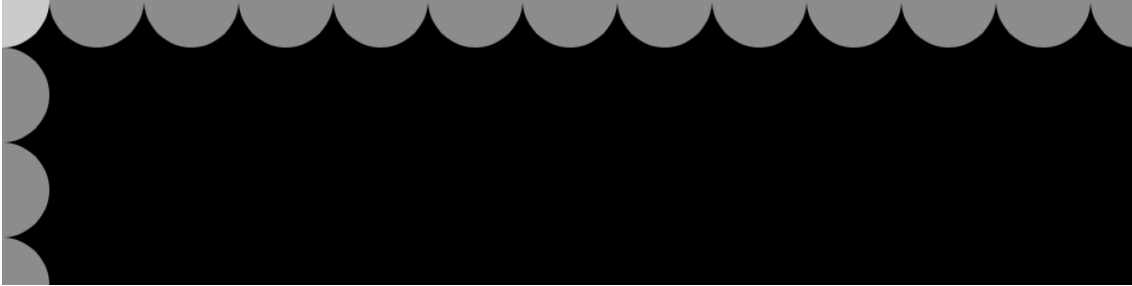
Quando um forloop é inserido dentro de outro, o número de repetições é multiplicado. Primeiro, vejamos um pequeno exemplo e depois o detalharemos no [Exemplo 4-11](#) :



```
function setup() {  
  createCanvas(480, 120);  
  noStroke();  
}  
  
function draw() {  
  background(0);  
  for (var y = 0; y <= height; y += 40) {  
    for (var x = 0; x <= width; x += 40) {  
      fill(255, 140);  
      ellipse(x, y, 40, 40);  
    }  
  }  
}
```

Exemplo 4-11: Linhas e Colunas

Nissopor exemplo, os forloops são adjacentes, em vez de um incorporado dentro do outro. O resultado mostra que um forloop desenha uma coluna de 4 círculos e o outro desenha uma linha de 13 círculos:



```
function setup() {
  createCanvas(480, 120);
  noStroke();
}

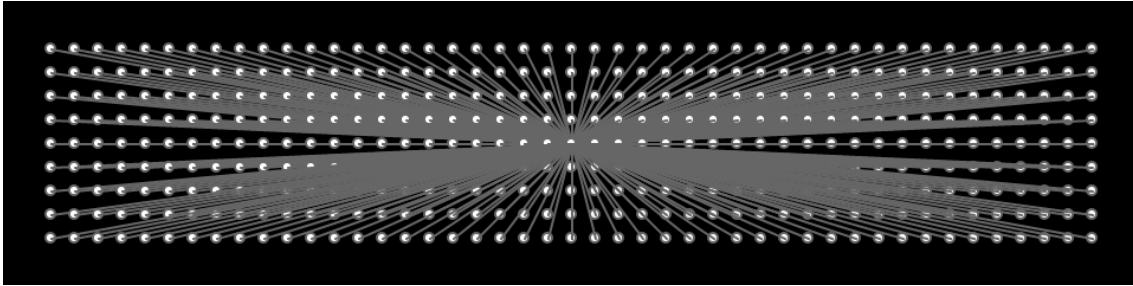
function draw() {
  background(0);
  for (var y = 0; y < height+45; y += 40) {
    fill(255, 140);
    ellipse(0, y, 40, 40);
  }
  for (var x = 0; x < width+45; x += 40) {
    fill(255, 140);
    ellipse(x, 0, 40, 40);
  }
}
```

Quando um desses forloops é colocado dentro do outro, como no [Exemplo 4-10](#), as 4 repetições do primeiro loop são compostas com as 13 do segundo para executar o código dentro do bloco incorporado 52 vezes ($4 \times 13 = 52$).

[O Exemplo 4-10](#) é uma boa base para explorar muitos tipos de padrões visuais repetidos. Os exemplos a seguir mostram algumas maneiras pelas quais ele pode ser estendido, mas esta é apenas uma pequena amostra do que está acontecendo.possível.

Exemplo 4-12: Pinos e Linhas

Neste exemplo, oo código desenha uma linha de cada ponto da grade até o centro da tela:

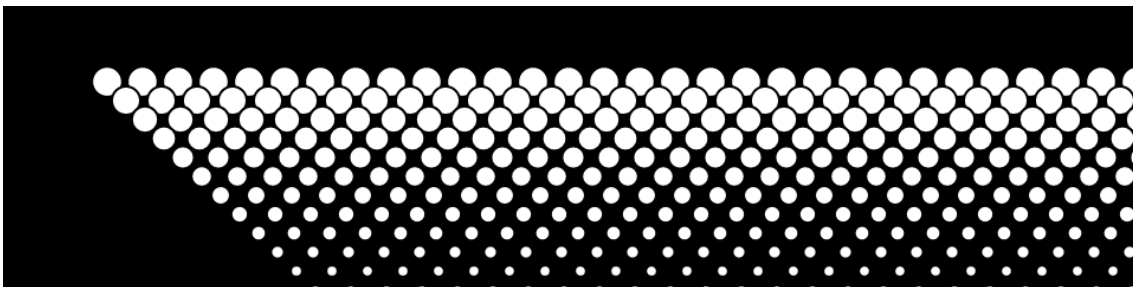


```
function setup() {
  createCanvas(480, 120);
  fill(255);
  stroke(102);
}

function draw() {
  background(0);
  for (var y = 20; y <= height-20; y += 10) {
    for (var x = 20; x <= width-20; x += 10) {
      ellipse(x, y, 4, 4);
      // Draw a line to the center of the display
      line(x, y, 240, 60);
    }
  }
}
```

Exemplo 4-13: Pontos de meio-tom

Neste exemplo, as elipses diminuem a cada nova linha e são movidas para a direita adicionando a coordenada y à coordenada x :

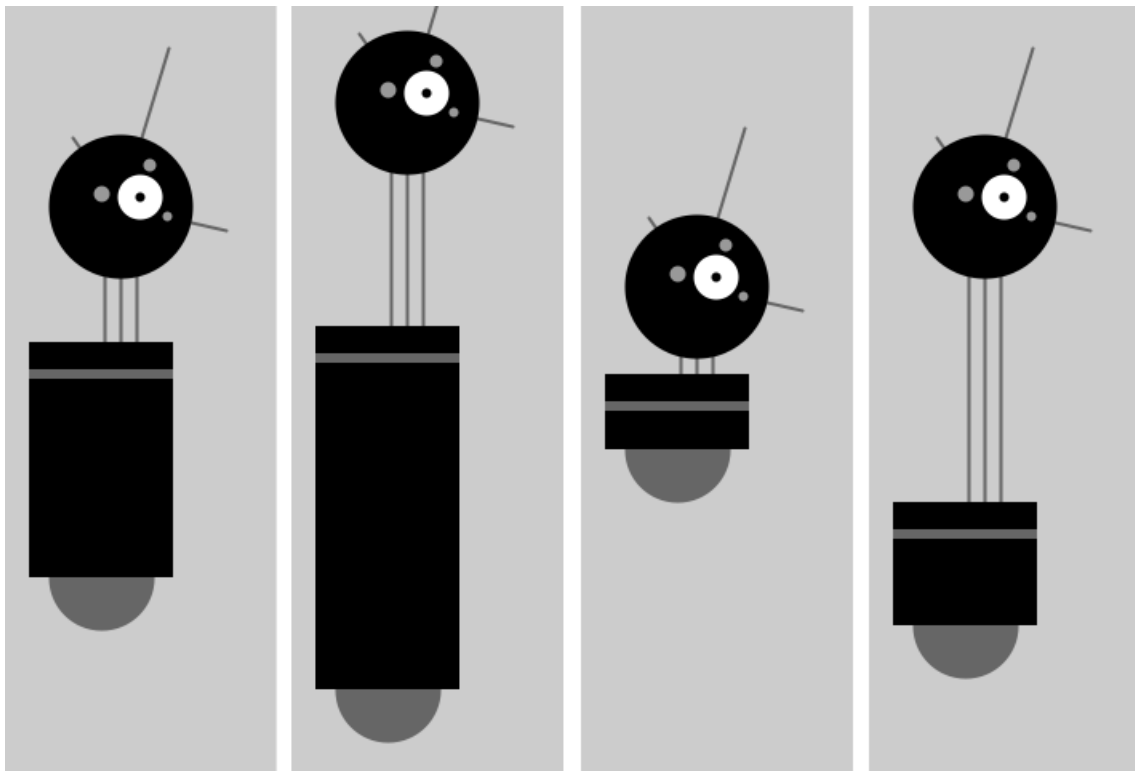


```
function setup() {
  createCanvas(480, 120);
}

function draw() {
  background(0);
  for (var y = 32; y <= height; y += 8) {
    for (var x = 12; x <= width; x += 15) {
      ellipse(x + y, y, 16 - y/10.0, 16 - y/10.0);
    }
  }
}
```

}

Robô 2: Variáveis



Oas variáveis introduzidas neste programa fazem com que o código pareça mais difícil do que o do Robô 1 (veja [“Robô 1: Desenhar”](#)), mas agora é muito mais fácil de modificar, porque os números que dependem uns dos outros estão em um único local. Por exemplo, o pescoço é desenhado com base na `neckHeight` variável. O grupo de variáveis no topo do código controla os aspectos do robô que queremos alterar: localização, altura do corpo e altura do pescoço. Você pode ver algumas das variações possíveis na figura; da esquerda para a direita, aqui estão os valores que lhes correspondem:

<code>y = 390</code>	<code>y = 460</code>	<code>y = 310</code>	<code>y = 420</code>
altura do corpo = 180	altura do corpo = 260	altura do corpo = 80	altura do corpo = 110
altura do pescoço = 40	altura do pescoço = 95	altura do pescoço = 10	altura do pescoço = 140

Ao alterar seu próprio código para usar variáveis em vez de números, planeje as alterações cuidadosamente e faça as modificações em etapas curtas. Por exemplo, quando este programa foi escrito, cada variável foi criada uma de cada vez para minimizar a complexidade da transição. Depois que uma variável foi adicionada e o código foi executado para garantir que estava funcionando, a próxima variável foi adicionada:

```

var x = 60;          // x coordinate
var y = 420;        // y coordinate
var bodyHeight = 110; // Body height
var neckHeight = 140; // Neck height
var radius = 45;
var ny = y - bodyHeight - neckHeight - radius; // Neck Y

function setup() {
  createCanvas(170, 480);
  strokeWeight(2);
  ellipseMode(RADIUS);
}

function draw() {
  background(204);

  // Neck
  stroke(102);
  line(x+2, y-bodyHeight, x+2, ny);
  line(x+12, y-bodyHeight, x+12, ny);
  line(x+22, y-bodyHeight, x+22, ny);

  // Antennae
  line(x+12, ny, x-18, ny-43);
  line(x+12, ny, x+42, ny-99);
  line(x+12, ny, x+78, ny+15);

  // Body
  noStroke();
  fill(102);
  ellipse(x, y-33, 33, 33);
  fill(0);
  rect(x-45, y-bodyHeight, 90, bodyHeight-33);
  fill(102);
  rect(x-45, y-bodyHeight+17, 90, 6);

  // Head
  fill(0);
  ellipse(x+12, ny, radius, radius);
  fill(255);
  ellipse(x+24, ny-6, 14, 14);
  fill(0);
  ellipse(x+24, ny-6, 3, 3);
  fill(153);
  ellipse(x, ny-8, 5, 5);
  ellipse(x+30, ny-26, 4, 4);
  ellipse(x+41, ny+6, 3, 3);
}

```

Chapter 5. Response

Code that responds to input from the mouse, keyboard, and other devices depends on the program to run continuously. We first encountered the `setup()` and `draw()` functions in [Chapter 1](#). Now we will learn more about what they do and how to use them to react to input to the program.

Once and Forever

The code within the `draw()` block runs from top to bottom, then repeats until you quit the program by closing the window. Each trip through `draw()` is called a *frame*. (The default frame rate is 60 frames per second, but this can be changed.)

Example 5-1: The `draw()` Function

To see how `draw()` works, run this example:

```
function draw() {  
  // Displays the frame count to the console  
  print("I'm drawing");  
  print(frameCount);  
}
```

You'll see the following:

```
I'm drawing  
1  
I'm drawing  
2  
I'm drawing  
3  
...
```

In the previous example program, the `print()` functions write the text “I’m drawing” followed by the current frame count as counted by the special `frameCount` variable (1, 2, 3, ...). The text appears in the console in your browser.

Example 5-2: The `setup()` Function

To complement the looping `draw()` function, p5.js has the `setup()` function that runs just once when the program starts:

```
function setup() {  
  print("I'm starting");  
}
```

```
function draw() {  
  print("I'm running");  
}
```

When this code is run, the following is written to the console:

```
I'm starting  
I'm running  
I'm running  
I'm running  
...
```

The text “I’m running” continues to write to the console until the program is stopped.

In some browsers, rather than printing “I’m running” over and over, it will print once, then for each subsequent time, it will increment a number next to the printed line, representing the total number of times that line has been printed in a row.

In a typical program, the code inside `setup()` is used to define the starting values. The first line is usually the `createCanvas()` function, often followed by code to set the starting fill and stroke colors. (If you don’t include the `createCanvas()` function, the drawing canvas will be 100×100 pixels.)

Now you know how to use `setup()` and `draw()` in more detail, but this isn’t the whole story.

There’s one more location you’ve been putting code—you can also place global variables outside of `setup()` and `draw()`. This is clearer when we list the order in which the code is run:

1. Variables declared outside of `setup()` and `draw()` are created.
2. Code inside `setup()` is run once.
3. Code inside `draw()` is run continuously.

Example 5-3: `setup()`, Meet `draw()`

The following example puts it all together:

```
var x = 280;  
var y = -100;  
var diameter = 380;  
  
function setup() {  
  createCanvas(480, 120);
```



```

fill(102);
}

function draw() {
  background(204);
  ellipse(x, y, diameter, diameter);
}

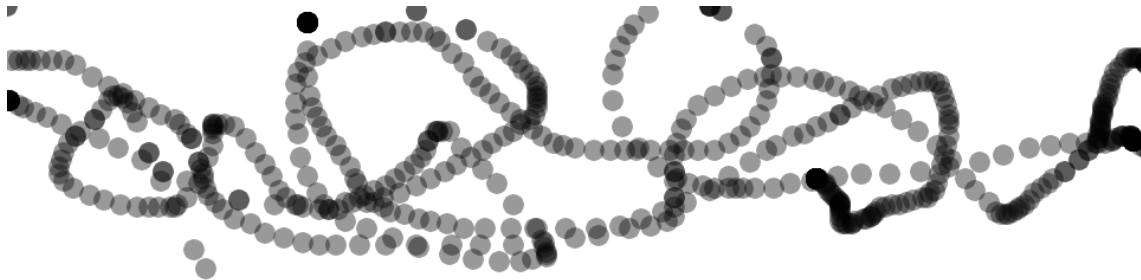
```

Follow

Because the code is running continuously, we can track the mouse position and use those numbers to move elements on screen.

Example 5-4: Track the Mouse

The `mouseX` variable stores the x coordinate, and the `mouseY` variable stores the y coordinate:



```

function setup() {
  createCanvas(480, 120);
  fill(0, 102);
  noStroke();
}

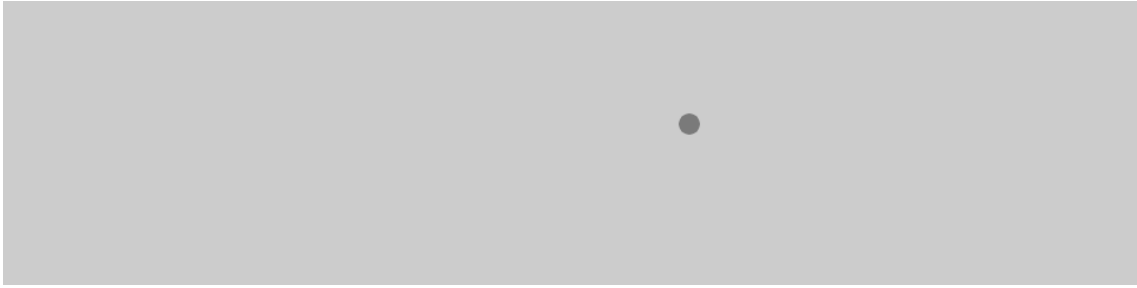
function draw() {
  ellipse(mouseX, mouseY, 9, 9);
}

```

In this example, each time the code in the `draw()` block is run, a new circle is drawn to the canvas. This image was made by moving the mouse around to control the circle's location. Because the fill is set to be partially transparent, denser black areas show where the mouse spent more time and where it moved slowly. The circles that are spaced farther apart show when the mouse was moving faster.

Example 5-5: The Dot Follows You

In this example, a new circle is added to the canvas each time the code in `draw()` is run. To refresh the screen and only display the newest circle, place a `background()` function at the beginning of `draw()` before the shape is drawn:

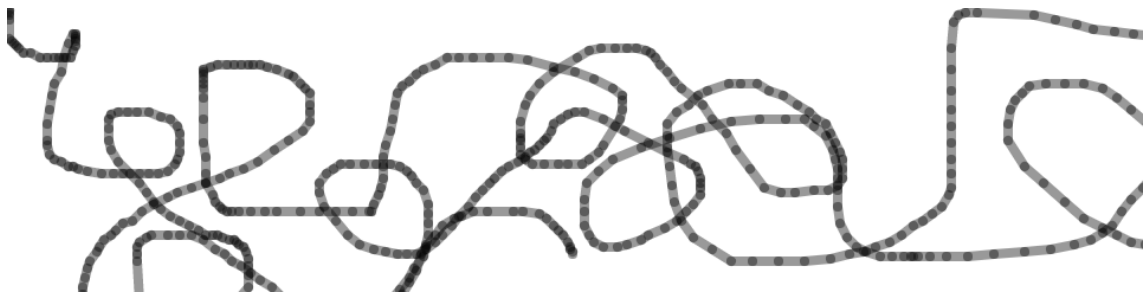


```
function setup() {  
  createCanvas(480, 120);  
  fill(0, 102);  
  noStroke();  
}  
  
function draw() {  
  background(204);  
  ellipse(mouseX, mouseY, 9, 9);  
}
```

The `background()` function clears the entire canvas, so be sure to always place it before other functions inside `draw()`; otherwise, the shapes drawn before it will be erased.

Example 5-6: Draw Continuously

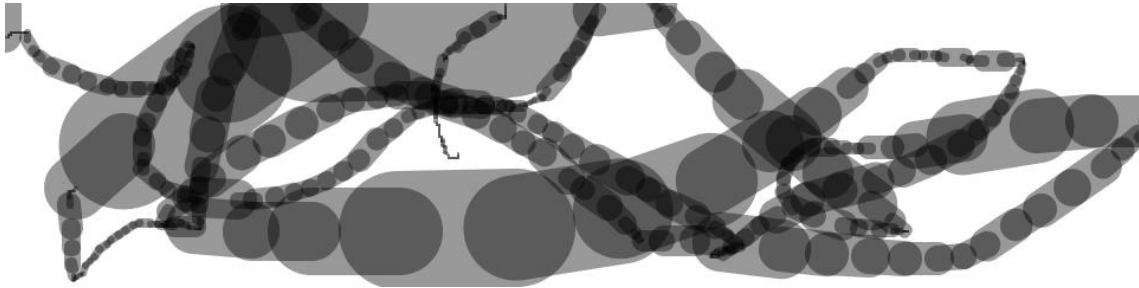
The `pmouseX` and `pmouseY` variables store the position of the mouse at the previous frame. Like `mouseX` and `mouseY`, these special variables are updated each time `draw()` runs. When combined, they can be used to draw continuous lines by connecting the current and most recent location:



```
function setup() {  
  createCanvas(480, 120);  
  strokeWeight(4);  
  stroke(0, 102);  
}  
  
function draw() {  
  line(mouseX, mouseY, pmouseX, pmouseY);  
}
```

Example 5-7: Set Thickness on the Fly

The `pmouseX` and `pmouseY` variables can also be used to calculate the speed of the mouse. This is done by measuring the distance between the current and most recent mouse location. If the mouse is moving slowly, the distance is small, but if the mouse starts moving faster, the distance grows. A function called `dist()` simplifies this calculation, as shown in the following example. Here, the speed of the mouse is used to set the thickness of the drawn line:



```
function setup() {  
  createCanvas(480, 120);  
  stroke(0, 102);  
}  
  
function draw() {  
  var weight = dist(mouseX, mouseY, pmouseX, pmouseY);  
  strokeWeight(weight);  
  line(mouseX, mouseY, pmouseX, pmouseY);  
}
```

Example 5-8: Easing Does It

In [Example 5-7](#), the values from the mouse are converted directly into positions on the screen. But sometimes you want the values to follow the mouse loosely—to lag behind to create a more fluid motion. This technique is called *easing*. With easing, there are two values: the current value and the value to move toward (see [Figure 5-1](#)). At each step in the program, the current value moves a little closer to the target value:

```
var x = 0;  
var easing = 0.01;  
  
function setup() {  
  createCanvas(220, 120);  
}  
  
function draw() {  
  var targetX = mouseX;  
  x += (targetX - x) * easing;  
  ellipse(x, 40, 12, 12);  
  print(targetX + " : " + x);  
}
```

The value of the x variable is always getting closer to targetX. The speed at which it catches up with targetX is set with the easing variable, a number between 0 and 1. A small value for easing causes more of a delay than a larger value. With an easing value of 1, there is no delay. When you run [Example 5-8](#), the actual values are shown in the console through the print() function. When moving the mouse, notice how the numbers are far apart, but when the mouse stops moving, the x value gets closer to targetX.

easing = 0.1



easing = 0.2



easing = 0.3



easing = 0.4

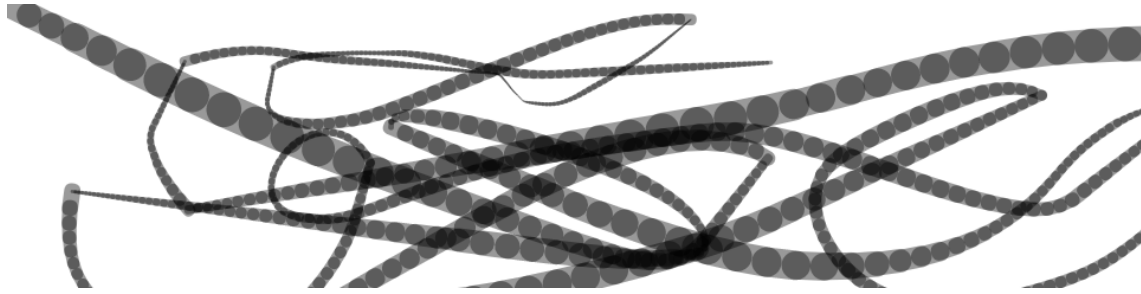


Figure 5-1. Easing changes the number of steps it takes to move from one place to another

All of the work in this example happens on the line that begins `x +=`. There, the difference between the target and current value is calculated, then multiplied by the easing variable and added to x to bring it closer to the target.

Example 5-9: Smooth Lines with Easing

In this example, the easing technique is applied to [Example 5-7](#). In comparison, the lines are more fluid:



```
var x = 0;
var y = 0;
var px = 0;
var py = 0;
var easing = 0.05;

function setup() {
  createCanvas(480, 120);
  stroke(0, 102);
}

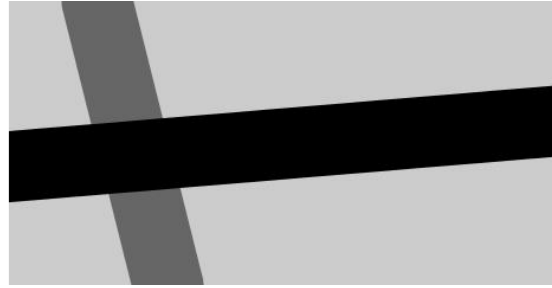
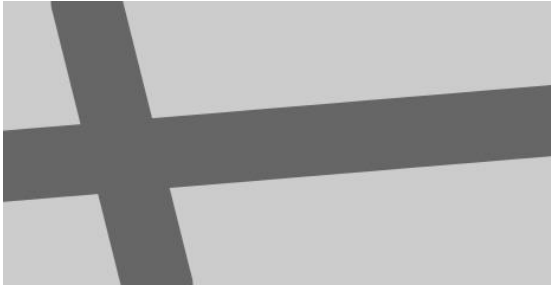
function draw() {
  var targetX = mouseX;
  x += (targetX - x) * easing;
  var targetY = mouseY;
  y += (targetY - y) * easing;
  var weight = dist(x, y, px, py);
  strokeWeight(weight);
  line(x, y, px, py);
  py = y;
  px = x;
}
```

Click

In addition to the location of the mouse, p5.js also keeps track of whether the mouse button is pressed. The `mouseIsPressed` variable has a different value when the mouse button is pressed and when it is not. The `mouseIsPressed` variable is called a boolean variable, which means that it has only two possible values: true and false. The value of `mouseIsPressed` is true when a button is pressed.

Example 5-10: Click the Mouse

The `mouseIsPressed` variable is used along with the if statement to determine when a line of code will run and when it won't. Try this example before we explain further:



```
function setup() {  
  createCanvas(240, 120);  
  strokeWeight(30);  
}  
  
function draw() {  
  background(204);  
  stroke(102);  
  line(40, 0, 70, height);  
  if (mouseIsPressed == true) {  
    stroke(0);  
  }  
  line(0, 70, width, 50);  
}
```

In this program, the code inside the if block runs only when a mouse button is pressed. When a button is not pressed, this code is ignored. Like the for loop discussed in [“Repetition”](#), the if also has a test that is evaluated to true or false:

```
if (test) {  
  statements  
}
```

When the test is true, the code inside the block is run; when the test is false, the code inside the block is not run. The computer determines whether the test is true or false by evaluating the expression inside the parentheses. (If you’d like to refresh your memory, [Example 4-6](#) more fully discusses relational expressions.)

The == symbol compares the values on the left and right to test whether they are equivalent. This == symbol is different from the assignment operator, the single = symbol. The == symbol asks, “Are these things equal?” and the = symbol sets the value of a variable.

NOTE

It’s a common mistake, even for experienced programmers, to write = in your code when you mean to write ==. p5.js won’t always warn you when you do this, so be careful.

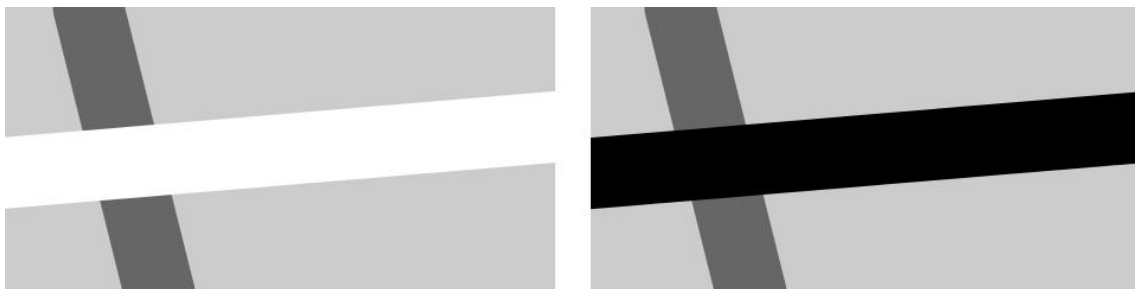
Alternatively, the test in draw() can be written like this:

```
if (mouseIsPressed) {
```

Boolean variables, including `mouseIsPressed`, don't need the explicit comparison with the `==` operator, because they will be only true or false.

Example 5-11: Detect When Not Clicked

A single if block gives you the choice of running some code or skipping it. You can extend an if block with an else block, allowing your program to choose between two options. The code inside the else block runs when the value of the if block test is false. For instance, the stroke color for a program can be white when the mouse button is not pressed, and can change to black when the button is pressed:



```
function setup() {  
  createCanvas(240, 120);  
  strokeWeight(30);  
}
```

```
function draw() {  
  background(204);  
  stroke(102);  
  line(40, 0, 70, height);  
  if (mouseIsPressed) {  
    stroke(0);  
  } else {  
    stroke(255);  
  }  
  line(0, 70, width, 50);  
}
```

Example 5-12: Multiple Mouse Buttons

p5.js also tracks which button is pressed if you have more than one button on your mouse. The `mouseButton` variable can be one of three values: `LEFT`, `CENTER`, or `RIGHT`. To test which button was pressed, the `==` operator is needed, as shown here:

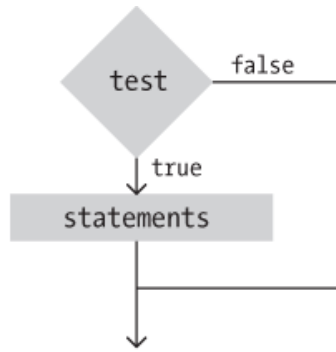


```
function setup() {  
  createCanvas(120, 120);  
  strokeWeight(30);  
}
```

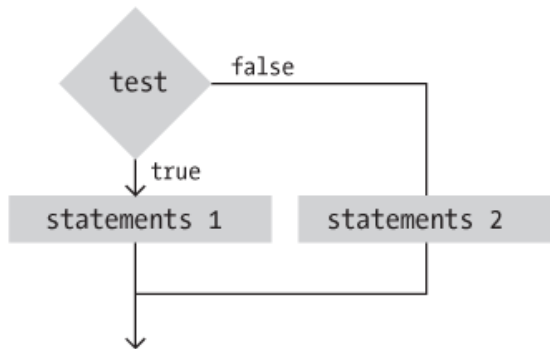
```
function draw() {  
  background(204);  
  stroke(102);  
  line(40, 0, 70, height);  
  if (mouseIsPressed) {  
    if (mouseButton == LEFT) {  
      stroke(255);  
    } else {  
      stroke(0);  
    }  
  }  
  line(0, 70, width, 50);  
}
```

A program can have many more if and else structures (see [Figure 5-2](#)) than those found in these short examples. They can be chained together into a long series with each testing for something different, and if blocks can be embedded inside of other if blocks to make more complex decisions.


```
if (test) {  
    statements  
}
```



```
if (test) {  
    statements 1  
} else {  
    statements 2  
}
```



```
if (test 1) {  
    statements 1  
} else if (test 2) {  
    statements 2  
}
```

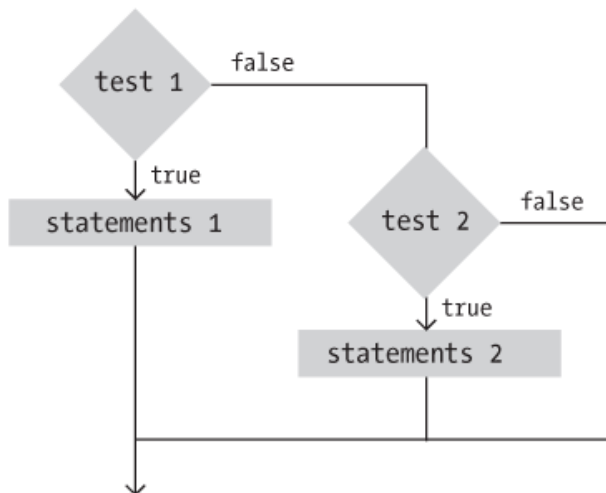


Figure 5-2. The if and else structure makes decisions about which blocks of code to run

Location

An if structure can be used with the mouseX and mouseY values to determine the location of the cursor within the window.

Example 5-13: Find the Cursor

For instance, this example tests to see whether the cursor is on the left or right side of a line and then moves the line toward the cursor:



```
var x;
var offset = 10;

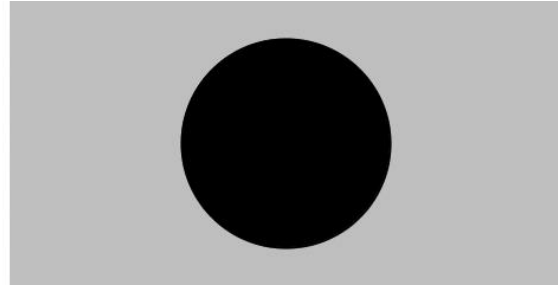
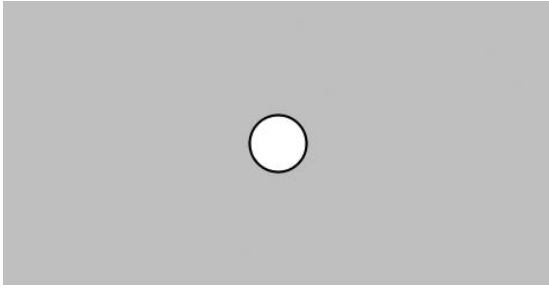
function setup() {
  createCanvas(240, 120);
  x = width/2;
}

function draw() {
  background(204);
  if (mouseX > x) {
    x += 0.5;
    offset = -10;
  }
  if (mouseX < x) {
    x -= 0.5;
    offset = 10;
  }
  // Draw arrow left or right depending on "offset" value
  line(x, 0, x, height);
  line(mouseX, mouseY, mouseX + offset, mouseY - 10);
  line(mouseX, mouseY, mouseX + offset, mouseY + 10);
  line(mouseX, mouseY, mouseX + offset*3, mouseY);
}
```

To write programs that have graphical user interfaces (buttons, checkboxes, scrollbars, etc.), we need to write code that knows when the cursor is within an enclosed area of the screen. The following two examples introduce how to check whether the cursor is inside a circle and a rectangle. The code is written in a modular way with variables, so it can be used to check for *any* circle and rectangle by changing the values.

Example 5-14: The Bounds of a Circle

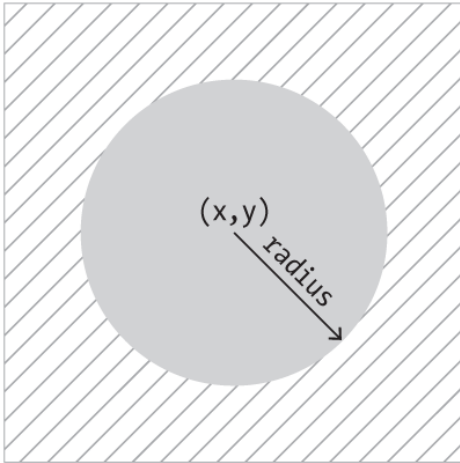
For the circle test, we use the `dist()` function to get the distance from the center of the circle to the cursor, then we test to see if that distance is less than the radius of the circle (see [Figure 5-3](#)). If it is, we know we're inside. In this example, when the cursor is within the area of the circle, its size increases:



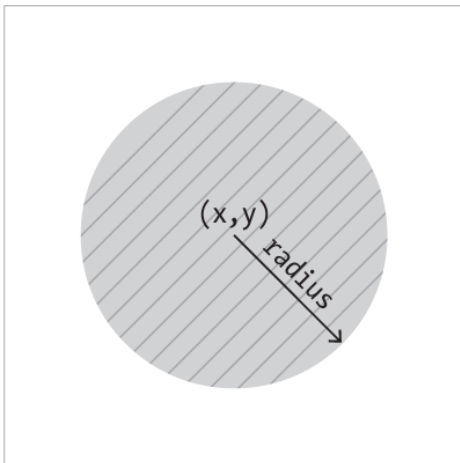
```
var x = 120;  
var y = 60;  
var radius = 12;
```

```
function setup() {  
  createCanvas(240, 120);  
  ellipseMode(RADIUS);  
}
```

```
function draw() {  
  background(204);  
  var d = dist(mouseX, mouseY, x, y);  
  if (d < radius) {  
    radius++;  
    fill(0);  
  } else {  
    fill(255);  
  }  
  ellipse(x, y, radius, radius);  
}
```



$$\text{dist}(x, y, \text{mouseX}, \text{mouseY}) > \text{radius}$$

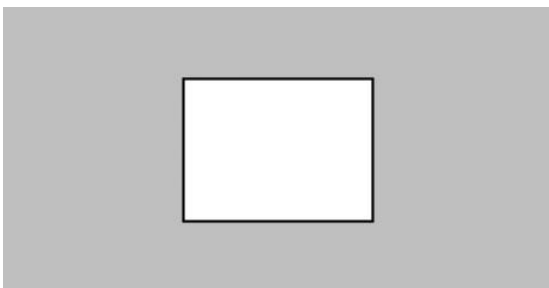


$$\text{dist}(x, y, \text{mouseX}, \text{mouseY}) < \text{radius}$$

Figure 5-3. Circle rollover test. When the distance between the mouse and the circle is less than the radius, the mouse is inside the circle.

Example 5-15: The Bounds of a Rectangle

We use another approach to test whether the cursor is inside a rectangle. We make four separate tests to check if the cursor is on the correct side of each edge of the rectangle, then we compare each test and if they are all true, we know the cursor is inside. This is illustrated in [Figure 5-4](#). Each step is simple, but it looks complicated when it's all put together:



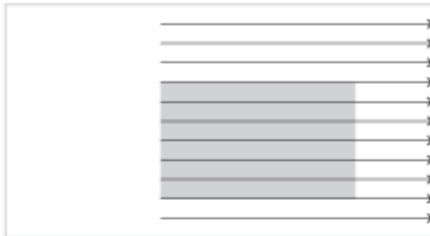
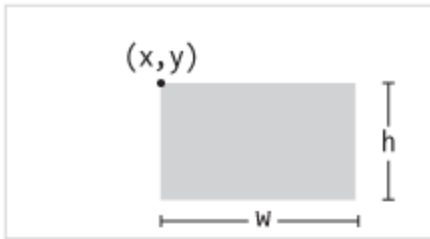
```
var x = 80;
```

```
var y = 30;
var w = 80;
var h = 60;

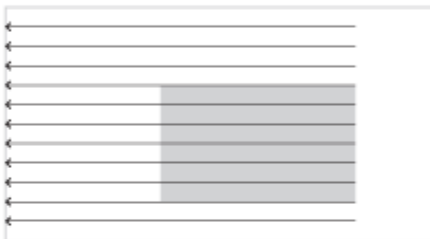
function setup() {
  createCanvas(240, 120);
}

function draw() {
  background(204);
  if ((mouseX > x) && (mouseX < x+w) &&
      (mouseY > y) && (mouseY < y+h)) {
    fill(0);
  }
  else {
    fill(255);
  }
  rect(x, y, w, h);
}
```

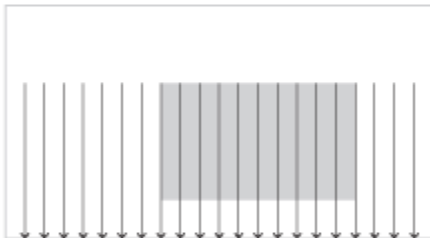
The test in the if statement is a little more complicated than we've seen. Four individual tests (e.g., `mouseX > x`) are combined with the logical AND operator, the `&&` symbol, to ensure that every relational expression in the sequence is true. If one of them is false, the entire test is false and the fill color won't be set to black.



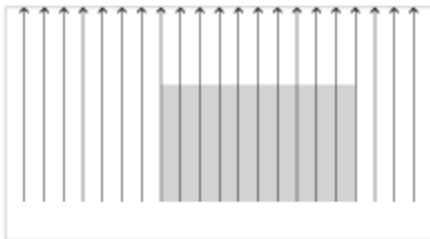
$\text{mouseX} > x$



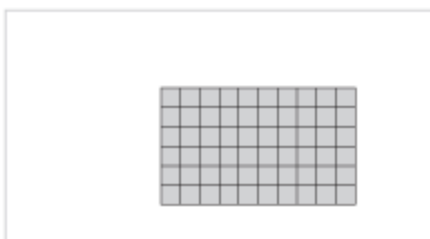
$\text{mouseX} < x + w$



$\text{mouseY} > y$



$\text{mouseY} < y + h$



$(\text{mouseX} > x) \ \&\& \ (\text{mouseX} < x+w) \ \&\& \ (\text{mouseY} > y) \ \&\& \ (\text{mouseY} < y+h)$

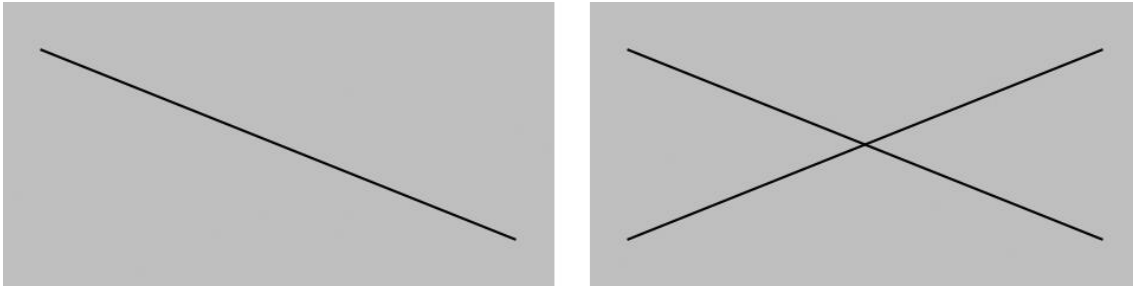
Figure 5-4. Rectangle rollover test. When all four tests are combined and true, the cursor is inside the rectangle.

Type

p5.js keeps track of when any key on a keyboard is pressed, as well as the last key pressed. Like the `mouseIsPressed` variable, the `keyIsPressed` variable is true when any key is pressed, and false when no keys are pressed.

Example 5-16: Tap a Key

In this example, the second line is drawn only when a key is pressed:



```
function setup() {  
  createCanvas(240, 120);  
}  
  
function draw() {  
  background(204);  
  line(20, 20, 220, 100);  
  if (keyIsPressed) {  
    line(220, 20, 20, 100);  
  }  
}
```

The `key` variable stores the most recent key that has been pressed. Unlike the boolean variable `keyIsPressed`, which reverts to false each time a key is released, the `key` variable keeps its value until the next key is pressed. The following example uses the value of `key` to draw the character to the screen. Each time a new key is pressed, the value updates and a new character draws. Some keys, like Shift and Alt, don't have a visible character, so when you press them, nothing is drawn.

Example 5-17: Draw Some Letters

This example introduces the `textSize()` function to set the size of the letters, the `textAlign()` function to center the text on its *x* coordinate, and the `text()` function to draw the letter. These functions are discussed in more detail in [“Fonts”](#).



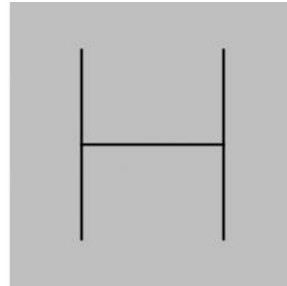
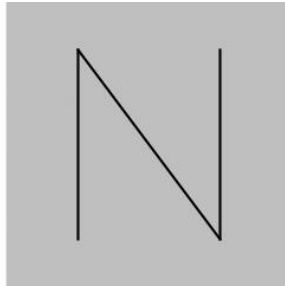
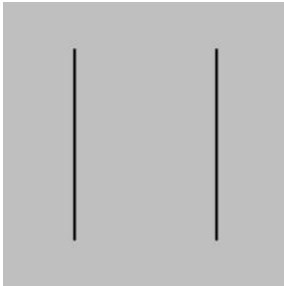
```
function setup() {  
  createCanvas(120, 120);  
  textSize(64);  
  textAlign(CENTER);  
  fill(255);  
}
```

```
function draw() {  
  background(0);  
  text(key, 60, 80);  
}
```

By using an if structure, we can test to see whether a specific key is pressed and choose to draw something on screen in response.

Example 5-18: Check for Specific Keys

In this example, we test for an H or N to be typed. We use the comparison operator, the == symbol, to see if the key value is equal to the characters we're looking for:



```
function setup() {  
  createCanvas(120, 120);  
}
```

```
function draw() {  
  background(204);  
  if (keyIsPressed) {  
    if ((key == 'h') || (key == 'H')) {  
      line(30, 60, 90, 60);  
    }  
    if ((key == 'n') || (key == 'N')) {  
      line(30, 20, 90, 100);  
    }  
  }  
}
```



```
}  
line(30, 20, 30, 100);  
line(90, 20, 90, 100);  
}
```

When we watch for H or N to be pressed, we need to check for both the lowercase and uppercase letters in the event that someone hits the Shift key or has the Caps Lock set. We combine the two tests together with a logical OR, the `||` symbol. If we translate the second if statement in this example into plain language, it says, “If the ‘h’ key is pressed OR the ‘H’ key is pressed.” Unlike with the logical AND (the `&&` symbol), only one of these expressions need be true for the entire test to be true.

Some keys are more difficult to detect, because they aren’t tied to a particular letter. Keys like Shift, Alt, and the arrow keys are coded. We check the code with the `keyCode` variable to see which key it is. The most frequently used `keyCode` values are `ALT`, `CONTROL`, and `SHIFT`, as well as the arrow keys, `UP_ARROW`, `DOWN_ARROW`, `LEFT_ARROW`, and `RIGHT_ARROW`.

Example 5-19: Move with Arrow Keys

The following example shows how to check for the left or right arrow keys to move a rectangle:

```
var x = 215;  
  
function setup() {  
  createCanvas(480, 120);  
}  
  
function draw() {  
  if (keyIsPressed) {  
    if (keyCode == LEFT_ARROW) {  
      x--;  
    }  
    else if (keyCode == RIGHT_ARROW) {  
      x++;  
    }  
  }  
  rect(x, 45, 50, 50);  
}
```

Touch

For devices that support it, p5.js keeps track of whether the screen is touched, and the location. Like the `mouseIsPressed` variable, the `touchIsDown` variable is true when the screen is touched, and false when it is not.

Example 5-20: Touch the Screen

In this example, the second line is drawn only when the screen is touched:

```
function setup() {
  createCanvas(240, 120);
}

function draw() {
  background(204);
  line(20, 20, 220, 100);
  if (touchIsDown) {
    line(220, 20, 20, 100);
  }
}
```

Like the mouseX and mouseY variables, the touchX and touchY variables store the x and y coordinates of the point where the screen is being touched.

Example 5-21: Track the Finger

In this example, a new circle is added to the canvas each time the code in draw() is run. To refresh the screen and only display the newest circle, place a background() function at the beginning of draw() before the shape is drawn:

```
function setup() {
  createCanvas(480, 120);
  fill(0, 102);
  noStroke();
}

function draw() {
  ellipse(touchX, touchY, 15, 15);
}
```

Map

The numbers that are created by the mouse and keyboard often need to be modified to be useful within a program. For instance, if a sketch is 1920 pixels wide and the mouseX values are used to set the color of the background, the range of 0 to 1920 for mouseX might need to move into a range of 0 to 255 to better control the color. This transformation can be done with an equation or with a function called map().

Example 5-22: Map Values to a Range

In this example, the location of two lines are controlled with the mouseX variable. The gray line is synchronized to the cursor position, but the black line stays closer to the center of the screen to move further away from the white line at the left and right edges:



```
function setup() {
  createCanvas(240, 120);
  strokeWeight(12);
}

function draw() {
  background(204);
  stroke(102);
  line(mouseX, 0, mouseX, height); // Gray line
  stroke(0);
  var mx = mouseX/2 + 60;
  line(mx, 0, mx, height); // Black line
}
```

The `map()` function is a more general way to make this type of change. It converts a variable from one range of numbers to another. The first parameter is the variable to be converted, the second and third parameters are the low and high values of that variable, and the fourth and fifth parameters are the desired low and high values. The `map()` function hides the math behind the conversion.

Example 5-23: Map with the `map()` Function

This example rewrites [Example 5-22](#) using `map()`:

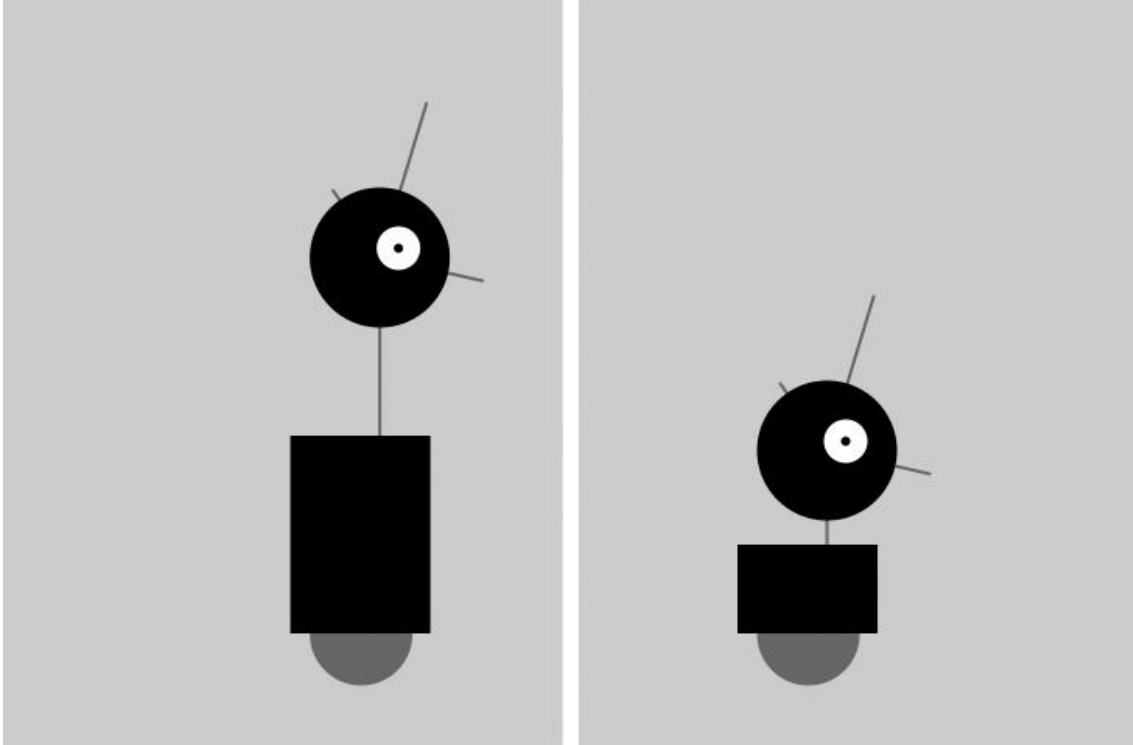
```
function setup() {
  createCanvas(240, 120);
  strokeWeight(12);
}

function draw() {
  background(204);
  stroke(255);
  line(120, 60, mouseX, mouseY); // White line
  stroke(0);
  var mx = map(mouseX, 0, width, 60, 180);
  line(120, 60, mx, mouseY); // Black line
}
```

The `map()` function makes the code easy to read, because the minimum and maximum values are clearly written as the parameters. In this example, `mouseX` values between 0

and width are converted to a number from 60 (when mouseX is 0) up to 180 (when mouseX is width). You'll find the useful `map()` function in many examples throughout this book.

Robot 3: Response



This program uses the variables introduced in Robot 2 (see [“Robot 2: Variables”](#)) and makes it possible to change them while the program runs so that the shapes respond to the mouse. The code inside the `draw()` block runs many times each second. At each frame, the variables defined in the program change in response to the `mouseX` and `mouseIsPressed` variables.

The `mouseX` value controls the position of the robot with an easing technique so that movements are less instantaneous and therefore feel more natural. When a mouse button is pressed, the values of `neckHeight` and `bodyHeight` change to make the robot short:

```
var x = 60;      // x coordinate
var y = 440;    // y coordinate
var radius = 45; // Head radius
var bodyHeight = 160; // Body height
var neckHeight = 70; // Neck height

var easing = 0.04;

function setup() {
  createCanvas(360, 480);
  strokeWeight(2);
  ellipseMode(RADIUS);
```

```

}

function draw() {

  var targetX = mouseX;
  x += (targetX - x) * easing;

  if (mouseIsPressed) {
    neckHeight = 16;
    bodyHeight = 90;
  } else {
    neckHeight = 70;
    bodyHeight = 160;
  }

  var neckY = y - bodyHeight - neckHeight - radius;

  background(204);

  // Neck
  stroke(102);
  line(x+12, y-bodyHeight, x+12, neckY);

  // Antennae
  line(x+12, neckY, x-18, neckY-43);
  line(x+12, neckY, x+42, neckY-99);
  line(x+12, neckY, x+78, neckY+15);

  // Body
  noStroke();
  fill(102);
  ellipse(x, y-33, 33, 33);
  fill(0);
  rect(x-45, y-bodyHeight, 90, bodyHeight-33);

  // Head
  fill(0);
  ellipse(x+12, neckY, radius, radius);
  fill(255);
  ellipse(x+24, neckY-6, 14, 14);
  fill(0);
  ellipse(x+24, neckY-6, 3, 3);

```

Chapter 6. Translate, Rotate, Scale

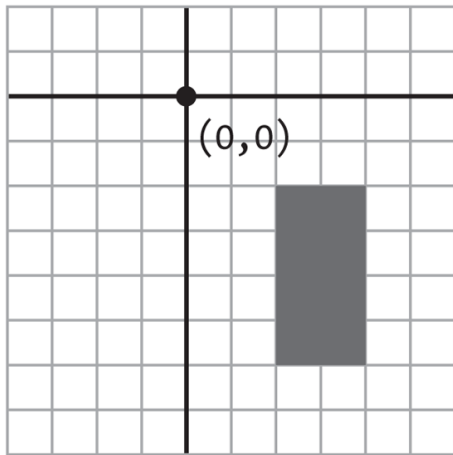
An alternative technique for positioning and moving things on screen is to change the screen coordinate system. For example, you can move a shape 50 pixels to the right, or you can move the location of coordinate (0,0) 50 pixels to the right—the visual result on screen is the same.

By modifying the default coordinate system, we can create different *transformations* including *translation*, *rotation*, and *scaling*.

Translate

Working with transformations can be tricky, but the `translate()` function is the most straightforward, so we'll start with that. As [Figure 6-1](#) shows, this function can shift the coordinate system left, right, up, and down.

```
translate(40, 20);  
rect(20, 20, 20, 40);
```



```
translate(60, 70);  
rect(20, 20, 20, 40);
```

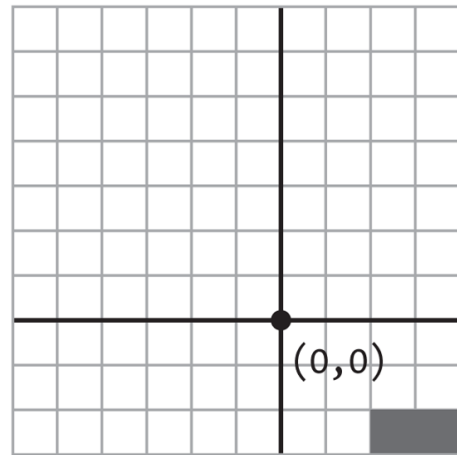
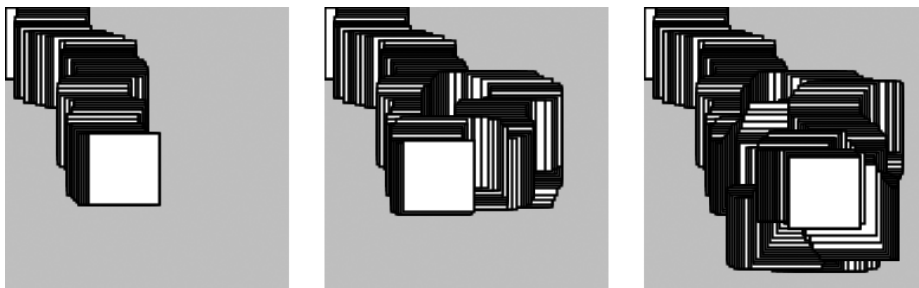


Figure 6-1. Translating the coordinates

Example 6-1: Translating Location

In this example, notice that the rectangle is drawn at coordinate (0,0), but it is moved around on the canvas, because it is affected by `translate()`:



```
function setup() {  
  createCanvas(120, 120);  
  background(204);  
}
```

```
function draw() {  
  translate(mouseX, mouseY);
```

```
rect(0, 0, 30, 30);  
}
```

The `translate()` function sets the (0,0) coordinate of the screen to the mouse location (`mouseX` and `mouseY`). Each time the `draw()` block repeats, the `rect()` is drawn at the new origin, derived from the current mouse location.

Example 6-2: Multiple Translations

After a transformation is made, it is applied to all drawing functions that follow. Notice what happens when a second `translate` function is added to control a second rectangle:



```
function setup() {  
  createCanvas(120, 120);  
  background(204);  
}  
  
function draw() {  
  translate(mouseX, mouseY);  
  rect(0, 0, 30, 30);  
  translate(35, 10);  
  rect(0, 0, 15, 15);  
}
```

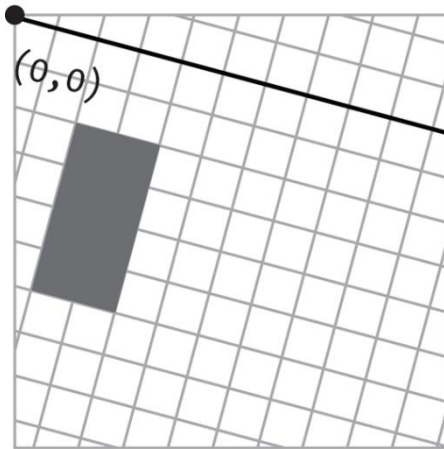
The values for the `translate()` functions are added together. The smaller rectangle was translated the amount of `mouseX + 35` and `mouseY + 10`. The x and y coordinates for both rectangles are (0,0), but the `translate()` functions move them to other positions on the canvas.

However, even though the transformations accumulate within the `draw()` block, they are reset each time `draw()` starts again at the top.

Rotate

The `rotate()` function rotates the coordinate system. It has one parameter, which is the angle (in radians) to rotate. It always rotates relative to (0,0), known as rotating around the *origin*. [Figure 3-2](#) shows the radians angle values. [Figure 6-2](#) shows the difference between rotating with positive and negative numbers.

```
rotate(PI/12.0);
rect(20, 20, 20, 40);
```



```
rotate(-PI/3);
rect(20, 20, 20, 40);
```

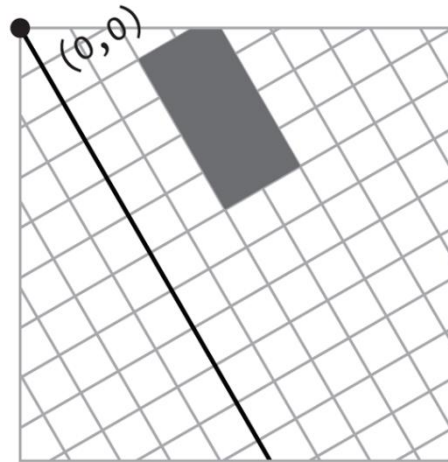
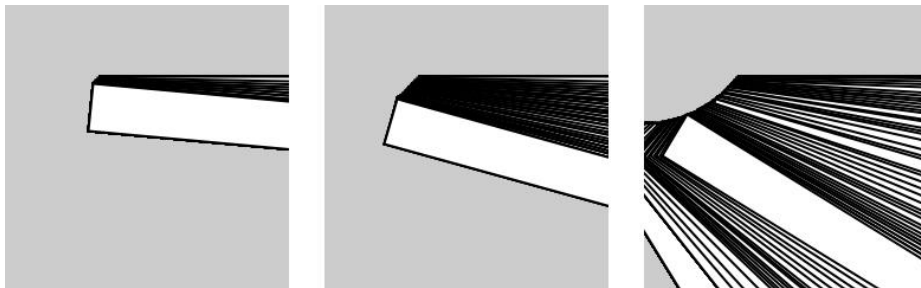


Figure 6-2. Rotating the coordinates

Example 6-3: Corner Rotation

To rotate a shape, first define the rotation angle with `rotate()`, then draw the shape. In this sketch, the parameter to `rotate` (`mouseX / 100.0`) will be between 0 and 1.2 to define the rotation angle because `mouseX` will be between 0 and 120, the width of the canvas as defined in `createCanvas()`:

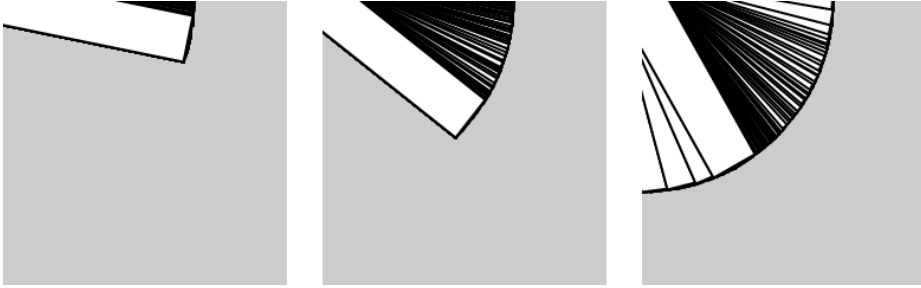


```
function setup() {
  createCanvas(120, 120);
  background(204);
}
```

```
function draw() {
  rotate(mouseX / 100.0);
  rect(40, 30, 160, 20);
}
```

Example 6-4: Center Rotation

To rotate a shape around its own center, it must be drawn with coordinate (0,0) in the middle. In this example, because the shape is 160 wide and 20 high as defined in `rect()`, it is drawn at the coordinate (-80, -10) to place (0,0) at the center of the shape:



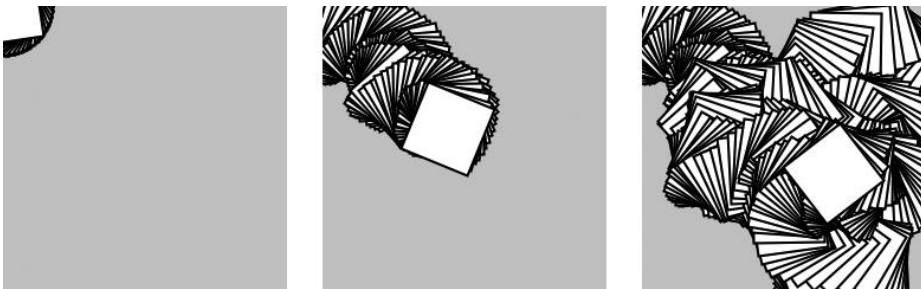
```
function setup() {
  createCanvas(120, 120);
  background(204);
}
```

```
function draw() {
  rotate(mouseX / 100.0);
  rect(-80, -10, 160, 20);
}
```

The previous pair of examples show how to rotate around coordinate (0,0), but what about other possibilities? You can use the `translate()` and `rotate()` functions for more control. When they are combined, the order in which they appear affects the result. If the coordinate system is first moved and then rotated, that is different than first rotating the coordinate system, then moving it.

Example 6-5: Translation, Then Rotation

To spin a shape around its center point at a place on screen away from the origin, first use `translate()` to move to the location where you'd like the shape, then call `rotate()`, and then draw the shape with its center at coordinate (0,0):



```
var angle = 0.0;
```

```
function setup() {
  createCanvas(120, 120);
  background(204);
}
```

```
function draw() {
  translate(mouseX, mouseY);
```

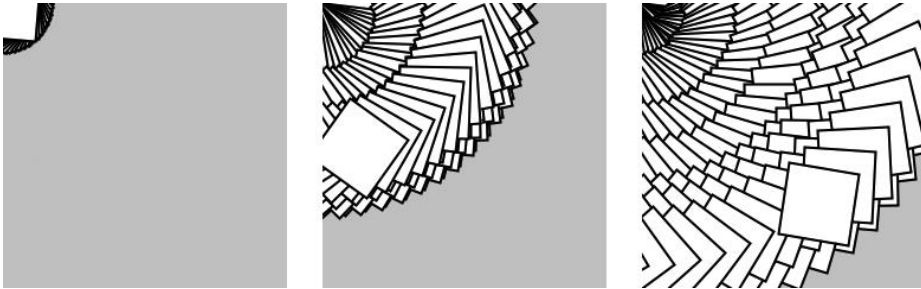
```

rotate(angle);
rect(-15, -15, 30, 30);
angle += 0.1;
}

```

Example 6-6: Rotation, Then Translation

The following example is identical to [Example 6-5](#), except that `translate()` and `rotate()` are reversed. The shape now rotates around the upper-left corner of the canvas, with the distance from the corner set by `translate()`:



```

var angle = 0.0;

function setup() {
  createCanvas(120, 120);
  background(204);
}

function draw() {
  rotate(angle);
  translate(mouseX, mouseY);
  rect(-15, -15, 30, 30);
  angle += 0.1;
}

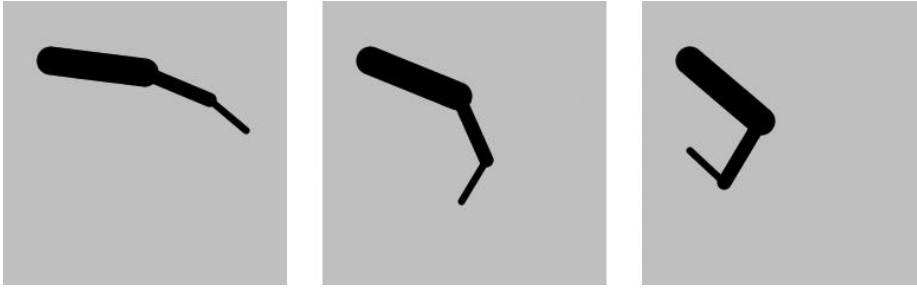
```

NOTE

You can also use the `rectMode()`, `ellipseMode()` and `imageMode()` functions to make it easier to draw shapes from their center. You can read about these functions in the *p5.js Reference*.

Example 6-7: An Articulating Arm

In this example, we've put together a series of `translate()` and `rotate()` functions to create a linked arm that bends back and forth. Each `translate()` further moves the position of the lines, and each `rotate()` adds to the previous rotation to bend more:



```

var angle = 0.0;
var angleDirection = 1;
var speed = 0.005;

function setup() {
  createCanvas(120, 120);
}

function draw() {
  background(204);
  translate(20, 25); // Move to start position
  rotate(angle);
  strokeWeight(12);
  line(0, 0, 40, 0);
  translate(40, 0); // Move to next joint
  rotate(angle * 2.0);
  strokeWeight(6);
  line(0, 0, 30, 0);
  translate(30, 0); // Move to next joint
  rotate(angle * 2.5);
  strokeWeight(3);
  line(0, 0, 20, 0);

  angle += speed * angleDirection;
  if ((angle > QUARTER_PI) || (angle < 0)) {
    angleDirection *= -1;
  }
}

```

The `angle` variable grows from 0 to `QUARTER_PI` (one quarter of the value of pi), then decreases until it is less than zero, then the cycle repeats. The value of the `angleDirection` variable is always 1 or -1 to make the value of `angle` correspondingly increase or decrease.

Scale

The `scale()` function stretches the coordinates on the canvas. Because the coordinates expand or contract as the scale changes, everything drawn to the canvas increases or decreases in dimension. The amount to scale is written as decimal percentages. Therefore, the parameter 1.5 to `scale()` is 150% and 3 is 300% ([Figure 6-3](#)).

```
scale(1.5);  
rect(20, 20, 20, 40);
```



```
scale(3);  
rect(20, 20, 20, 40);
```

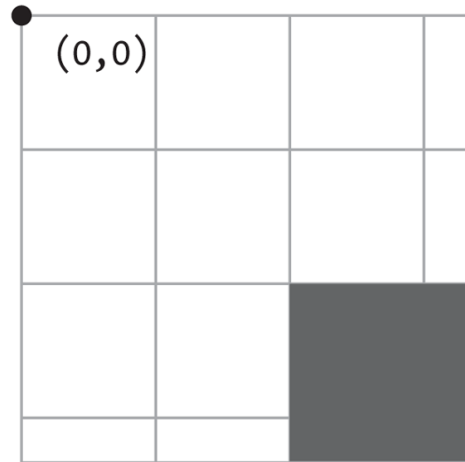
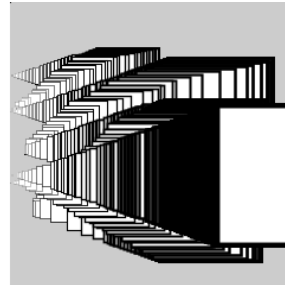
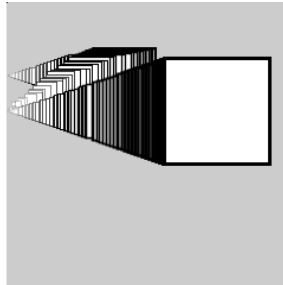
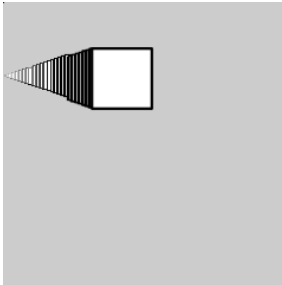


Figure 6-3. Scaling the coordinates

Example 6-8: Scaling

Like `rotate()`, the `scale()` function transforms from the origin. Therefore, as with `rotate()`, to scale a shape from its center, translate to its location, scale, and then draw with the center at coordinate (0,0):



```
function setup() {  
  createCanvas(120, 120);  
  background(204);  
}
```

```
function draw() {  
  translate(mouseX, mouseY);  
  scale(mouseX / 60.0);  
  rect(-15, -15, 30, 30);  
}
```

Example 6-9: Keeping Strokes Consistent

From the thick lines in [Example 6-8](#), you can see how the `scale()` function affects the stroke weight. To maintain a consistent stroke weight as a shape scales, divide the desired stroke weight by the scalar value:

```
function setup() {
  createCanvas(120, 120);
  background(204);
}

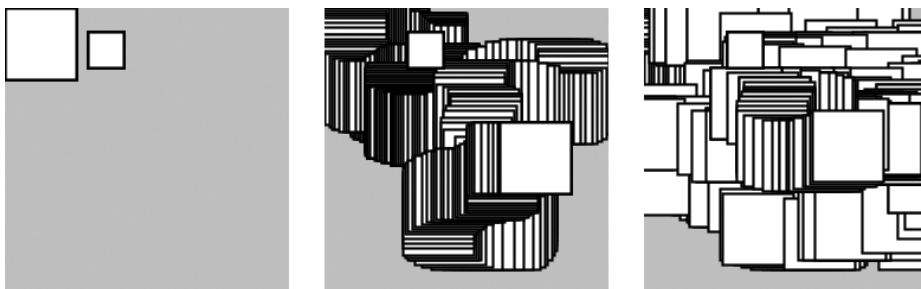
function draw() {
  translate(mouseX, mouseY);
  var scalar = mouseX / 60.0;
  scale(scalar);
  strokeWeight(1.0 / scalar);
  rect(-15, -15, 30, 30);
}
```

Push and Pop

To isolate the effects of transformations so they don't affect later functions, use the `push()` and `pop()` functions. When `push()` is run, it saves a copy of the current coordinate system and then restores that system after `pop()`. This is useful when transformations are needed for one shape, but not wanted for another.

Example 6-10: Isolating Transformations

In this example, the smaller rectangle always draws in the same position because the `translate(mouseX, mouseY)` is cancelled by the `pop()`:



```
function setup() {
  createCanvas(120, 120);
  background(204);
}

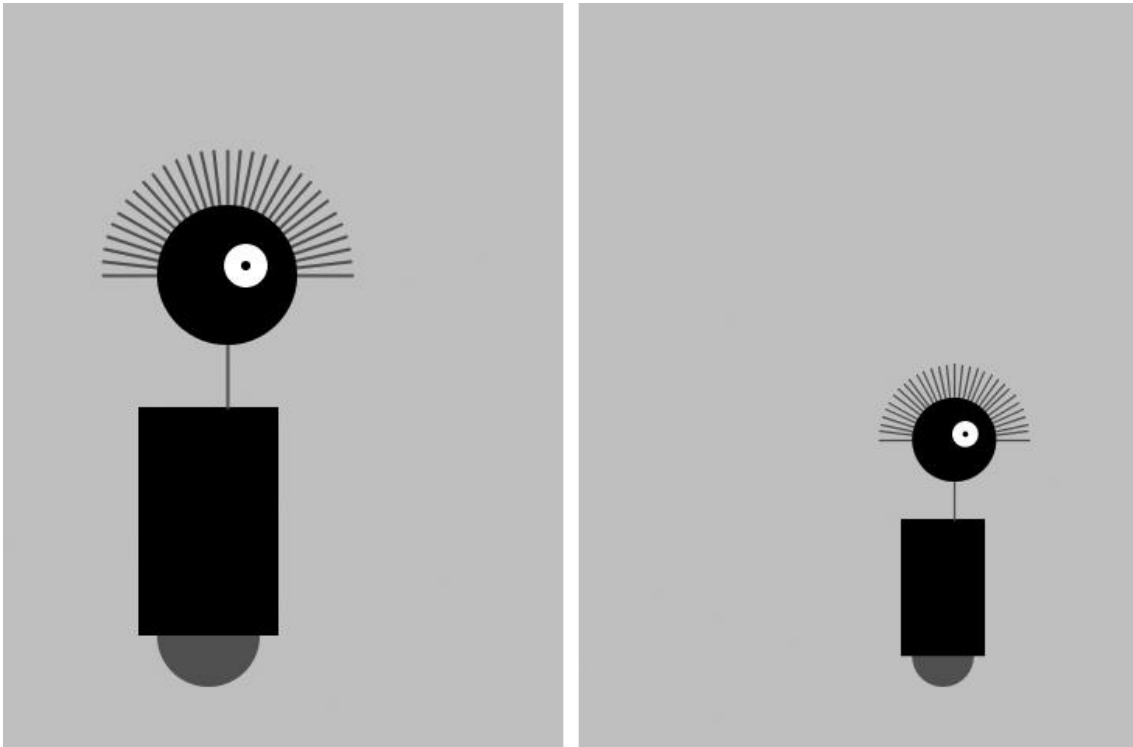
function draw() {
  push();
  translate(mouseX, mouseY);
  rect(0, 0, 30, 30);
  pop();
  translate(35, 10);
}
```

```
rect(0, 0, 15, 15);  
}
```

NOTE

The `push()` and `pop()` functions are always used in pairs. For every `push()`, you need to have a matching `pop()`.

Robot 4: Translate, Rotate, Scale



The `translate()`, `rotate()`, and `scale()` functions are all utilized in this modified robot sketch. In relation to [“Robot 3: Response”](#), `translate()` is used to make the code easier to read. Here, notice how the `x` value no longer needs to be added to each drawing function because the `translate()` moves everything. Similarly, the `scale()` function is used to set the dimensions for the entire robot. When the mouse is not pressed, the size is set to 60%, and when it is pressed, it goes to 100% in relation to the original coordinates. The `rotate()` function is used within a loop to draw a line, rotate it a little, then draw a second line, then rotate a little more, and so on until the loop has drawn 30 lines half-way around a circle to style a lovely head of robot hair:

```
var x = 60;      // x coordinate  
var y = 440;    // y coordinate  
var radius = 45; // Head radius  
var bodyHeight = 180; // Body height  
var neckHeight = 40; // Neck height  
  
var easing = 0.04;
```

```

function setup() {
  createCanvas(360, 480);
  strokeWeight(2);
  ellipseMode(RADIUS);
}

function draw() {

  var neckY = -1 * (bodyHeight + neckHeight + radius);

  background(204);

  translate(mouseX, y); // Move all to (mouseX, y)

  if (mouseIsPressed) {
    scale(1.0);
  } else {
    scale(0.6); // 60% size when mouse is pressed
  }

  // Body
  noStroke();
  fill(102);
  ellipse(0, -33, 33, 33);
  fill(0);
  rect(-45, -bodyHeight, 90, bodyHeight-33);

  // Neck
  stroke(102);
  line(12, -bodyHeight, 12, neckY);

  // Hair
  push();
  translate(12, neckY);
  var angle = -PI/30.0;
  for (var i = 0; i <= 30; i++) {
    line(80, 0, 0, 0);
    rotate(angle);
  }
  pop();

  // Head
  noStroke();
  fill(0);
  ellipse(12, neckY, radius, radius);
  fill(255);
  ellipse(24, neckY-6, 14, 14);
  fill(0);
  ellipse(24, neckY-6, 3, 3);

```

}

Chapter 7. Media

p5.js is capable of drawing more than simple lines and shapes. It's time to learn how to create images and text in our programs to extend the visual possibilities to photography, detailed diagrams, and diverse typefaces.

Before we do that, we first need to talk a little bit about servers. Up to this point, we've just been viewing the *index.html* file directly in the browser. This works fine for things like running simple animations. However, if you want to do things like load an external image file into your sketch, you might find that your browser doesn't allow this. If you look in the console, you may see an error containing the term *cross-origin*. For loading external files, you will need to run a *server*. A *server* is a program that works as a handler layer. It responds when you type a URL into the address bar, and *serves* the corresponding files to you for you to view.

There are several different ways to run servers.

Visit <https://github.com/processing/p5.js/wiki/Local-server> for instructions on how to run a server on Mac OS X, Windows, and Linux systems. Once you have that set up, you are ready to load media!

We've posted some media files online for you to use in this chapter's examples: <http://p5js.org/learn/books/media.zip>.

Download this file, unzip it to the desktop (or somewhere else convenient), and make a note of its location.

NOTE

To unzip on Mac OS X, just double-click the file, and it will create a folder named *media*. On Windows, double-click the *media.zip* file, which will open a new window. In that window, drag the *media* folder to the desktop.

Create a new sketch, and copy the *lunar.jpg* file from the *media* folder that you just unzipped into your *sketch* folder.

NOTE

On Windows and Mac OS X, extensions are hidden by default. It's a good idea to change that option so that you always see the full name of your files. On Mac OS X, select Preferences from the Finder menu, and then make sure "Show all filename extensions" is checked in the Advanced tab. On Windows, look for Folder Options, and set the option there.

Images

There are three steps to follow before you can draw an image to the screen:

1. Add the image to the sketch's folder.

2. Create a variable to store the image.
3. Load the image into the variable with `loadImage()`.

Example 7-1: Load an Image

In order to load an image, we will introduce a new function called `preload()`. The `preload()` function runs once before the `setup()` function runs. You should generally load your images and other media in `preload()` in order to ensure they are fully loaded before your program starts. We will discuss this in more depth later in the chapter.

After all three steps are done, you can draw the image to the screen with the `image()` function. The first parameter to `image()` specifies the image to draw; the second and third set the x and y coordinates:



```
var img;

function preload() {
  img = loadImage("lunar.jpg");
}

function setup() {
  createCanvas(480, 120);
}

function draw() {
  image(img, 0, 0);
}
```

Optional fourth and fifth parameters set the width and height to draw the image. If the fourth and fifth parameters are not used, the image is drawn at the size at which it was created.

These next examples show how to work with more than one image in the same program and how to resize an image.

Example 7-2: Load More Images

For this example, you'll need to add the `capsule.jpg` file (found in the `media` folder you downloaded) to your `sketch` folder:



```
var img1;  
var img2;  
  
function preload() {  
  img1 = loadImage("lunar.jpg");  
  img2 = loadImage("capsule.jpg");  
}  
  
function setup() {  
  createCanvas(480, 120);  
}  
  
function draw() {  
  image(img1, -120, 0);  
  image(img1, 130, 0, 240, 120);  
  image(img2, 300, 0, 240, 120);  
}
```

Example 7-3: Mousing Around with Images

When the mouseX and mouseY values are used as part of the fourth and fifth parameters of image(), the image size changes as the mouse moves:



```
var img;  
  
function preload() {  
  img = loadImage("lunar.jpg");  
}  
  
function setup() {  
  createCanvas(480, 120);  
}
```

```
function draw() {  
  background(0);  
  image(img, 0, 0, mouseX * 2, mouseY * 2);  
}
```

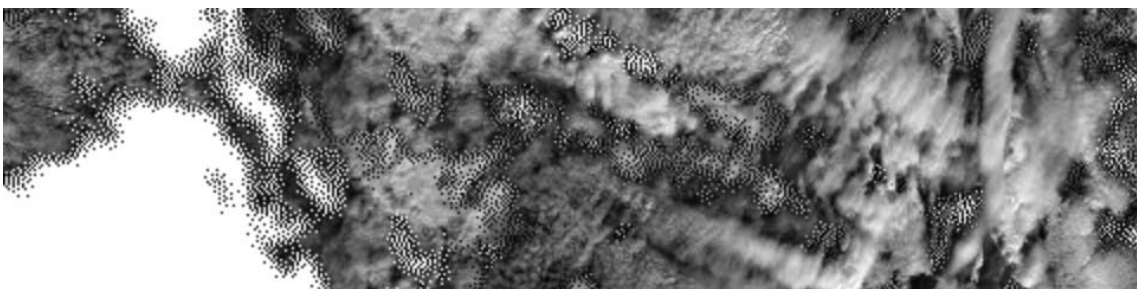
NOTE

When an image is displayed larger or smaller than its actual size, it may become distorted. Be careful to prepare your images at the sizes they will be used. When the display size of an image is changed with the `image()` function, the actual image in your *sketch* folder doesn't change.

p5.js can load and display raster images in the JPEG, PNG, and GIF formats, and vector images in the SVG format. You can convert images to the JPEG, PNG, GIF, and SVG formats using programs like GIMP, Photoshop, and Illustrator. Most digital cameras save JPEG images, but they usually need to be reduced in size before being used with p5.js. A typical digital camera creates an image that is several times larger than the drawing area of most p5.js sketches. Resizing these images before they are added to the *sketch* folder makes sketches load faster, run more efficiently, and can save disk space.

GIF, PNG, and SVG images support transparency, which means that pixels can be invisible or partially visible (recall the discussion of `color()` and alpha values in [Example 3-17](#)). GIF images have 1-bit transparency, which means that pixels are either fully opaque or fully transparent. PNG images have 8-bit transparency, meaning each pixel can have a variable level of opacity. The following examples use *clouds.gif* and *clouds.png* to show the differences between the file formats. These images are in the *media* folder that you downloaded previously. Be sure to add them to the sketch before trying each example.

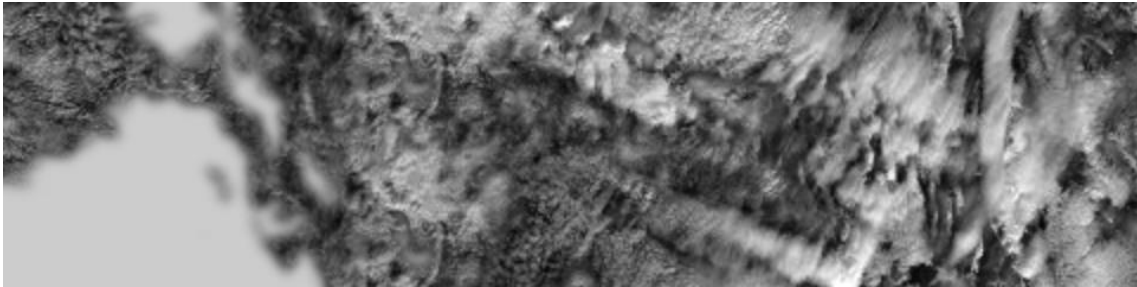
Example 7-4: Transparency with a GIF



```
var img;  
  
function preload() {  
  img = loadImage("clouds.gif");  
}  
  
function setup() {  
  createCanvas(480, 120);  
}
```

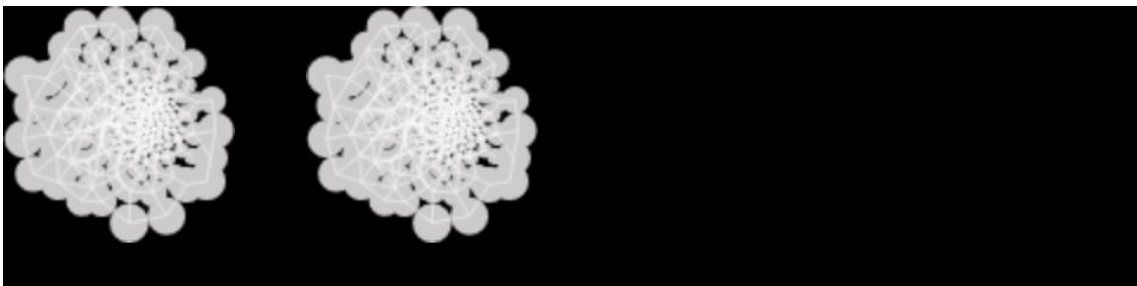
```
function draw() {  
  background(204);  
  image(img, 0, 0);  
  image(img, 0, mouseY * -1);  
}
```

Example 7-5: Transparency with a PNG



```
var img;  
  
function preload() {  
  img = loadImage("clouds.png");  
}  
  
function setup() {  
  createCanvas(480, 120);  
}  
  
function draw() {  
  background(204);  
  image(img, 0, 0);  
  image(img, 0, mouseY * -1);  
}
```

Example 7-6: Displaying an SVG Image



```
var img;  
  
function preload() {  
  img = loadImage("network.svg");  
}  
  
function setup() {
```

```
    createCanvas(480, 120);
  }

  function draw() {
    background(0);
    image(img, 0, 0);
    image(img, mouseX, 0);
  }
```

NOTE

Remember to include the appropriate file extension (*.gif*, *.jpg*, *.png*, or *.svg*) when you load the image. Also, be sure that the image name is typed exactly as it appears in the file, including the case of the letters.

Asynchronicity

Why do we need to load images in `preload()`? Why not use `setup()`? Up until this point, we've been assuming that our programs run from top to bottom, with each line completing before going on to the next one. Although this is generally true, when it comes to certain functions like loading images, your browser will begin the process of loading the image, but skip onto the next line before the image finishes loading! This is known as *asynchronicity*, or an *asynchronous function*. It's a little bit unexpected at first, but this ultimately allows your pages to load and run faster on the Web.

To see this more clearly, consider the following example. It is identical to [Example 7-1](#), except we use `loadImage()` in `setup()` instead of `preload()`.

Example 7-7: Demonstrating Asynchronicity



```
var img;

function setup() {
  createCanvas(480, 120);
  img = loadImage("lunar.jpg");
  noLoop();
}

function draw() {
  background(204);
  image(img, 0, 0);
}
```

```
}
```

When you run this program, you'll notice that the drawing canvas is gray with no image displayed. The sketch runs the `setup()` function first, then it runs the `draw()` function. At the `loadImage()` line, it begins to load the image, but continues on through the rest of `setup()` and on to `draw()` before the image has completely loaded. The `image()` function is unable to draw the not yet existing image.

To help with this issue, p5.js has the `preload()` function. Unlike `setup()`, `preload()` forces the program to wait until everything has loaded before moving on. It's best to only make load calls in `preload()`, and do all other setup in `setup()`.

Alternatively, instead of using `preload()`, you could use something called a *callback function*. A callback function is a function that is passed as an argument to a second function, and runs after the second function has completed. The following example illustrates this technique.

Example 7-8: Loading with a Callback

```
function setup() {  
  createCanvas(480, 120);  
  loadImage("lunar.jpg", drawImage);  
  noLoop();  
}  
  
function draw() {  
  background(200);  
}  
  
function drawImage(img) {  
  image(img, 0, 0);  
}
```

In this example, we add a second argument to `loadImage()`, which is the function we want to run after the load is complete. Once the image has loaded, the callback function `drawImage()` is automatically called, with one argument, the image that has just loaded.

There's no need to create a global variable to hold the image. The image is passed directly into the callback function, as the parameter name chosen in the function definition.

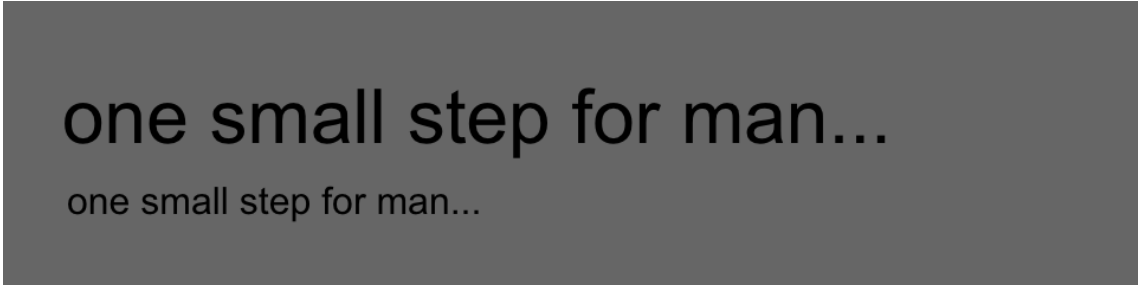
Fonts

p5.js can display text in many fonts other than the default. You can use any font already on your computer (these are called *system fonts*). Keep in mind that if you are sharing this on the Web, other people will also need to have the system font in order to see the text in the typeface you choose. There are a small number of fonts that *most* computers

and devices have; these include “Arial,” “Courier,” “Courier New,” “Georgia,” “Helvetica,” “Palatino,” “Times New Roman,” “Trebuchet MS,” and “Verdana.”

Example 7-9: Drawing with Fonts

You can use the `textFont()` function to set the current font. You can draw letters to the screen with the `text()` function, and you can change the size with `textSize()`:



one small step for man...
one small step for man...

```
function setup() {  
  createCanvas(480, 120);  
  textFont("Arial");  
}  
  
function draw() {  
  background(102);  
  textSize(32);  
  text("one small step for man...", 25, 60);  
  textSize(16);  
  text("one small step for man...", 27, 90);  
}
```

The first parameter to `text()` is the character(s) to draw to the screen. (Notice that the characters are enclosed within quotes.) The second and third parameters set the horizontal and vertical location. The location is relative to the baseline of the text (see [Figure 7-1](#)).



Figure 7-1. Typography coordinates

Example 7-10: Use a Webfont

If you don't want to be limited to this small list of fonts, you can use a webfont. Two websites that are good places to find webfonts with open licenses to use with p5.js are [GoogleFonts](#) and the [Open Font Library](#).

To use a webfont in your program, you'll need to link to it in your *index.html* file. When you choose a font from either of the aforementioned libraries, a snippet of code to add to your HTML file will be displayed. When you copy and paste this code anywhere in the <head> section of your HTML, your file will end up looking something like this:

```
<html>
<head>
<script type="text/javascript" src="../lib/p5.js"></script>
<script type="text/javascript" src="sketch.js"></script>
<link href="http://fonts.googleapis.com/css?family=Source+Code+Pro"
rel="stylesheet" type="text/css">
</head>
<body>
</body>
</html></pre>
```

Once you have linked the font in, you can use it with `textFont()` just like the system fonts:



one small step for man...
one small step for man...

```
function setup() {
  createCanvas(480, 120);
  textFont("Source Code Pro");
}

function draw() {
  background(102);
  textSize(28);
  text("one small step for man...", 25, 60);
  textSize(16);
  text("one small step for man...", 27, 90);
}
```

Example 7-11: Load a Custom Font

p5.js can also display text using TrueType (.*ttf*) and OpenType (.*otf*) fonts. For this introduction, we'll load the *SourceCodePro-Regular.ttf* TrueType font (included in the *media* folder that you downloaded earlier) from the *sketch* folder.

We're using the same font that we employed in [Example 7-10](#), but now the file is located in the *sketch* folder rather than loaded from somewhere else online. The output of the following program should look the same as [Example 7-10](#). Here are the steps you will follow to include and use a custom font in your program:

1. Add the font to the sketch's folder.
2. Create a variable to store the font.
3. Load the font into the variable with `loadFont()`.
4. Use the `textFont()` function to set the current font:
- 5.
6. `var font;`
- 7.
8. `function preload() {`
9. `font = loadFont("SourceCodePro-Regular.ttf");`
10. `}`
- 11.
12. `function setup() {`
13. `createCanvas(480, 120);`
14. `textFont(font);`
15. `}`
- 16.
17. `function draw() {`
18. `background(102);`
19. `textSize(28);`
20. `text("one small step for man...", 25, 60);`
21. `textSize(16);`
22. `text("one small step for man...", 27, 90);`
- `}`

Example 7-12: Set the Text Stroke and Fill

Just like shapes, the text is affected by both the `stroke()` and `fill()` functions. The following example outputs black text with a white outline:



one small step for man...
one small step for man...

```
function setup() {  
  createCanvas(480, 120);  
  textFont("Source Code Pro");
```

```

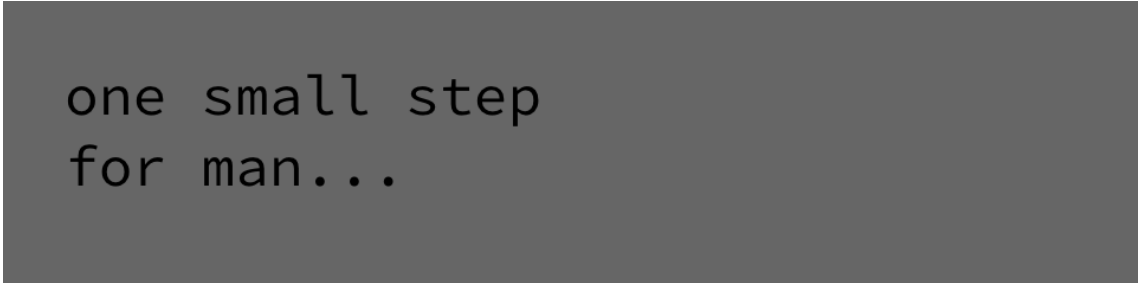
fill(0);
stroke(255);
}

function draw() {
  background(102);
  textSize(28);
  text("one small step for man...", 25, 60);
  textSize(16);
  text("one small step for man...", 27, 90);
}

```

Example 7-13: Draw Text in a Box

You can also set text to draw inside a box by adding fourth and fifth parameters that specify the width and height of the box:



one small step
for man...

```

function setup() {
  createCanvas(480, 120);
  textFont("Source Code Pro");
  textSize(24);
}

function draw() {
  background(102);
  text("one small step for man...", 26, 24, 240, 100);
}

```

Example 7-13: Store Text in a Variable

In the previous example, the words inside the text() function make the code difficult to read. We can store these words in a variable to make the code more modular. Here's a new version of the previous example that uses a variable:

```

var quote = "one small step for man...";

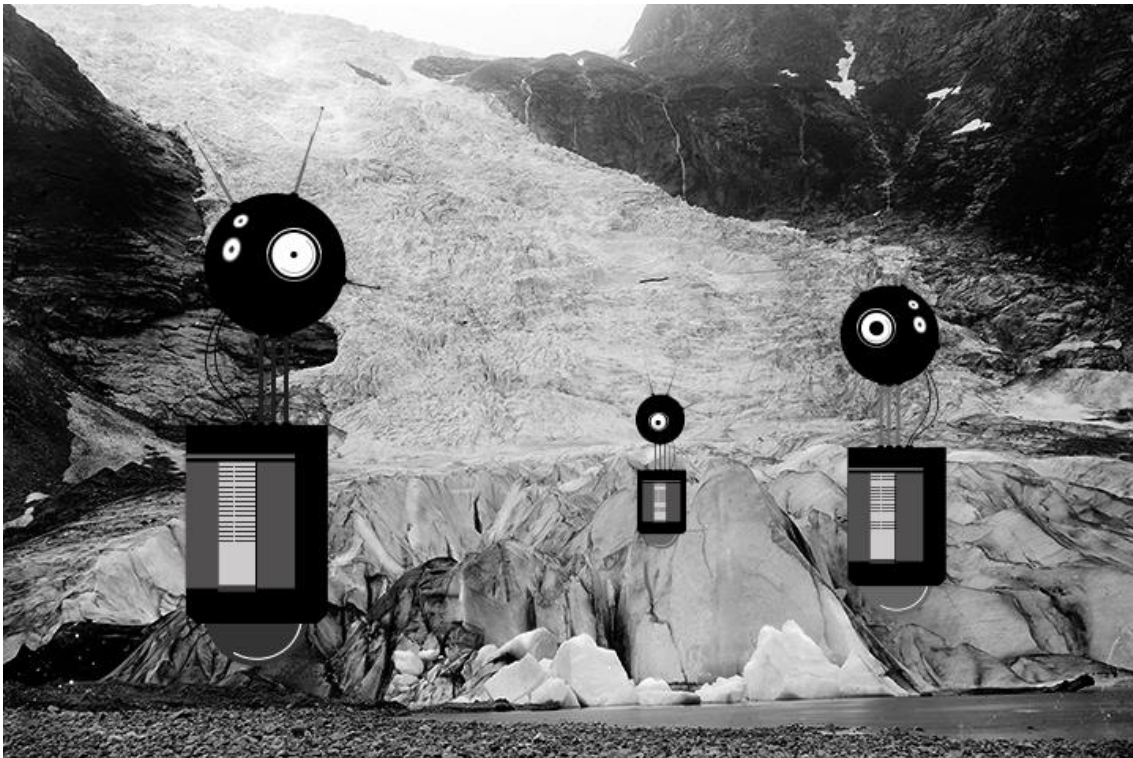
function setup() {
  createCanvas(480, 120);
  textFont("Source Code Pro");
  textSize(24);
}

```

```
}  
  
function draw() {  
  background(204);  
  text(quote, 26, 24, 240, 100);  
}
```

There's a set of additional functions that affect how letters are displayed on screen. They are explained, with examples, in the *Typography* category of the *p5.js Reference*.

Robot 5: Media



Unlike the robots created from lines and rectangles drawn in *p5.js* in the previous chapters, these robots were created with a vector drawing program. For some shapes, it's often easier to point and click in a software tool like *Inkscape* or *Illustrator* than to define the shapes with coordinates in code.

There's a trade-off to selecting one image creation technique over another. When shapes are defined in *p5.js*, there's more flexibility to modify them while the program is running. If the shapes are defined elsewhere and then loaded into *p5.js*, changes are limited to the position, angle, and size. When loading each robot from an *SVG* file, as this example shows, the variations featured in *Robot 2* (see [“Robot 2: Variables”](#)) are impossible.

Images can be loaded into a program to bring in visuals created in other programs or captured with a camera. With this image in the background, our robots are now exploring for life-forms in Norway at the dawn of the 20th century.

The SVG and PNG file used in this example can be downloaded from <http://p5js.org/learn/books/media.zip>:

```
var bot1;
var bot2;
var bot3;
var landscape;

var easing = 0.05;
var offset = 0;

// Preload the images
function preload() {
  bot1 = loadImage("robot1.svg");
  bot2 = loadImage("robot2.svg");
  bot3 = loadImage("robot3.svg");
  landscape = loadImage("alpine.png");
}

function setup() {
  createCanvas(720, 480);
}

function draw() {
  // Set the background to the "landscape" image; this image
  // must be the same width and height as the program
  background(landscape);

  // Set the left/right offset and apply easing to make
  // the transition smooth
  var targetOffset = map(mouseY, 0, height, -40, 40);
  offset += (targetOffset - offset) * easing;

  // Draw the left robot
  image(bot1, 85 + offset, 65);

  // Draw the right robot smaller and give it a smaller offset
  var smallerOffset = offset * 0.7;
  image(bot2, 510 + smallerOffset, 140, 78, 248);

  // Draw the smallest robot, give it a smaller offset
  smallerOffset *= -0.5;
  image(bot3, 410 + smallerOffset, 225, 39, 124);
}
```

Chapter 8. Motion

Like a flip book, animation on screen is created by drawing an image, then drawing a slightly different image, then another, and so on. The illusion of fluid motion is created

by *persistence of vision*. When a set of similar images is presented at a fast enough rate, our brains translate these images into motion.

Frames

To create smooth motion, p5.js tries to run the code inside `draw()` at 60 frames each second. A *frame* is one trip through the `draw()` function and the *frame rate* is how many frames are drawn each second. Therefore, a program that draws 60 frames each second means the program runs the entire code inside `draw()` 60 times each second.

Example 8-1: See the Frame Rate

To confirm the frame rate, we can use the browser console that we first learned to access in [Chapter 1](#). The `frameRate()` function tells you the current speed of your program. Open the console, run this program, and watch the values print out:

```
function draw() {  
  var fr = frameRate();  
  print(fr);  
}
```

Example 8-2: Set the Frame Rate

The `frameRate()` function can also change the speed at which the program runs. When there is no parameter passed in (like [Example 8-1](#)), it returns the current frame rate. However, if you call the `frameRate()` function with a parameter, it sets the frame rate to that value. To see the result, uncomment different versions of `frameRate()` in this example:

```
function setup() {  
  frameRate(30); // Thirty frames each second  
  //frameRate(12); // Twelve frames each second  
  //frameRate(2); // Two frames each second  
  //frameRate(0.5); // One frame every two seconds  
}  
  
function draw() {  
  var fr = frameRate();  
  print(fr);  
}
```

NOTE

p5.js *tries* to run the code at 60 frames each second, but if it takes longer than 1/60th of a second to run the `draw()` method, then the frame rate will decrease.

The `frameRate()` function specifies only the maximum frame rate, and the actual frame rate for any program depends on the computer that is running the code.

Speed and Direction

To create fluid motion examples, we create variables that store numbers and modify them a little bit each frame.

Example 8-3: Move a Shape

The following example moves a shape from left to right by updating the x variable:



```
var radius = 40;  
var x = -radius;  
var speed = 0.5;
```

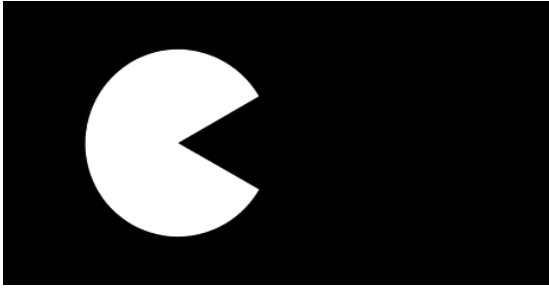
```
function setup() {  
  createCanvas(240, 120);  
  ellipseMode(RADIUS);  
}
```

```
function draw() {  
  background(0);  
  x += speed; // Increase the value of x  
  arc(x, 60, radius, radius, 0.52, 5.76);  
}
```

When you run this code, you'll notice the shape moves off the right of the screen when the value of the x variable is greater than the width of the window. The value of x continues to increase, but the shape is no longer visible.

Example 8-4: Wrap Around

There are many alternatives to this behavior, which you can choose according to your preference. First, we'll extend the code to show how to move the shape back to the left edge of the screen after it disappears off the right. In this case, picture the screen as a flattened cylinder, with the shape moving around the outside to return to its starting point:



```
var radius = 40;  
var x = -radius;  
var speed = 0.5;
```

```
function setup() {  
  createCanvas(240, 120);  
  ellipseMode(RADIUS);  
}
```

```
function draw() {  
  background(0);  
  x += speed; // Increase the value of x  
  if (x > width+radius) { // If the shape is off screen  
    x = -radius; // move to the left edge  
  }  
  arc(x, 60, radius, radius, 0.52, 5.76);  
}
```

On each trip through `draw()`, the code tests to see if the value of `x` has increased beyond the width of the screen (plus the radius of the shape). If it has, we set the value of `x` to a negative value, so that as it continues to increase, it will enter the screen from the left. See [Figure 8-1](#) for a diagram of how it works.

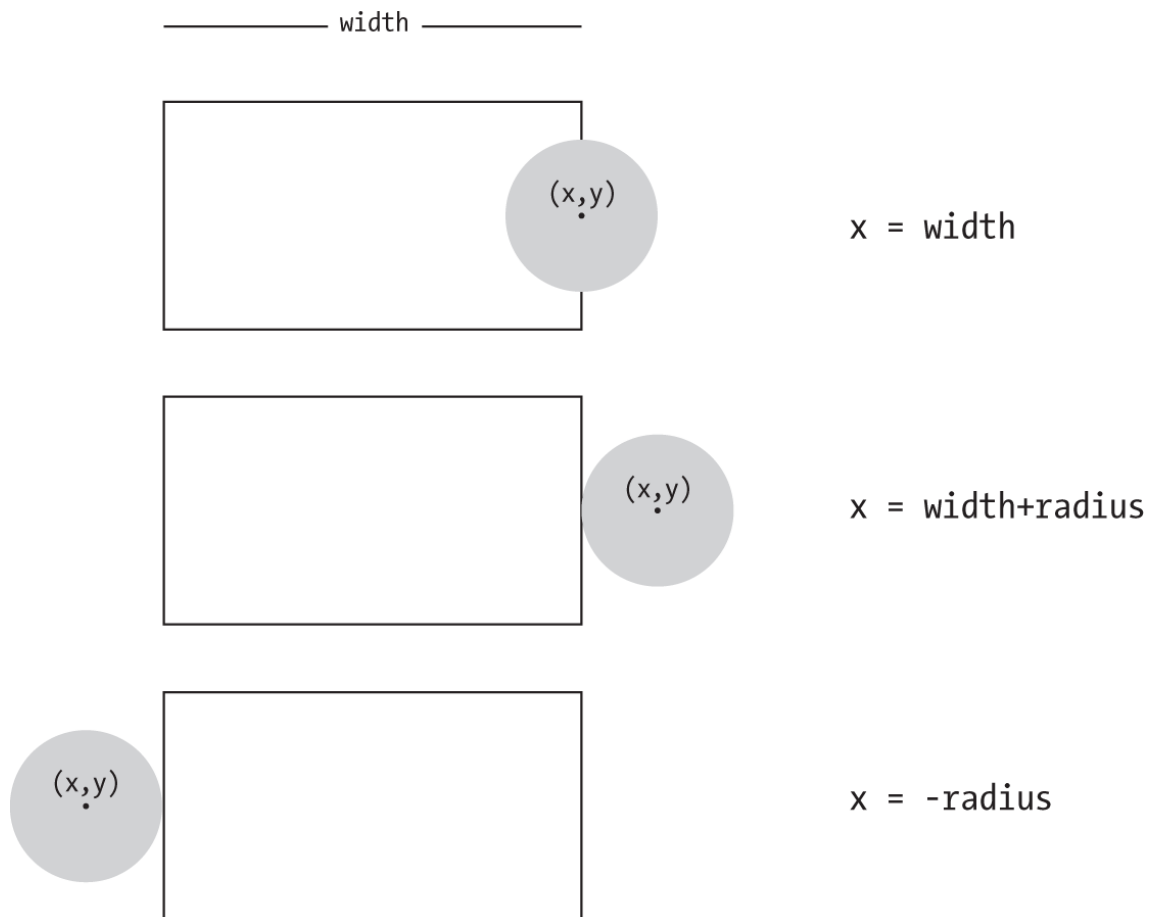
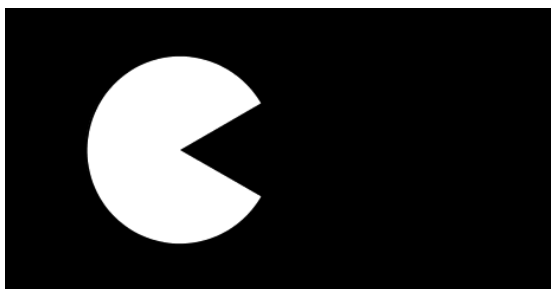


Figure 8-1. Testing for the edges of the window

Example 8-5: Bounce Off the Wall

In this example, we'll extend [Example 8-3](#) to have the shape change directions when it hits an edge, instead of wrapping around to the left. To make this happen, we add a new variable to store the direction of the shape. A direction value of 1 moves the shape to the right, and a value of -1 moves the shape to the left:



```
var radius = 40;
var x = 110;
var speed = 0.5;
var direction = 1;
```

```
function setup() {
```



```

createCanvas(240, 120);
ellipseMode(RADIUS);
}

function draw() {
  background(0);
  x += speed * direction;
  if ((x > width-radius) || (x < radius)) {
    direction = -direction; // Flip direction
  }
  if (direction == 1) {
    arc(x, 60, radius, radius, 0.52, 5.76); // Face right
  } else {
    arc(x, 60, radius, radius, 3.67, 8.9); // Face left
  }
}
}

```

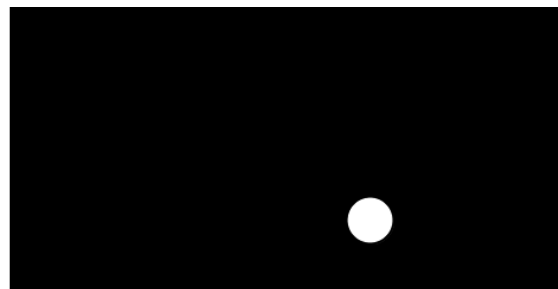
When the shape reaches an edge, this code flips the shape's direction by changing the sign of the direction variable. For example, if the direction variable is positive when the shape reaches an edge, the code flips it to negative.

Tweening

Sometimes you want to animate a shape to go from one point on screen to another. With a few lines of code, you can set up the start position and the stop position, then calculate the in-between (*tween*) positions at each frame.

Example 8-6: Calculate Tween Positions

To make this example code modular, we've created a group of variables at the top. Run the code a few times and change the values to see how this code can move a shape from any location to any other at a range of speeds. Change the step variable to alter the speed:



```

var startX = 20; // Initial x coordinate
var stopX = 160; // Final x coordinate
var startY = 30; // Initial y coordinate
var stopY = 80; // Final y coordinate
var x = startX; // Current x coordinate
var y = startY; // Current y coordinate
var step = 0.005; // createCanvas of each step (0.0 to 1.0)

```

```

var pct = 0.0; // Percentage traveled (0.0 to 1.0)

function setup() {
  createCanvas(240, 120);
}

function draw() {
  background(0);
  if (pct < 1.0) {
    x = startX + ((stopX-startX) * pct);
    y = startY + ((stopY-startX) * pct);
    pct += step;
  }
  ellipse(x, y, 20, 20);
}

```

Random

Unlike the smooth, linear motion common to computer graphics, motion in the physical world is usually idiosyncratic. For instance, think of a leaf floating to the ground, or an ant crawling over rough terrain. We can simulate the unpredictable qualities of the world by generating random numbers. The `random()` function calculates these values; we can set a range to tune the amount of disarray in a program.

Example 8-7: Generate Random Values

The following short example prints random values to the console, with the range limited by the position of the mouse:

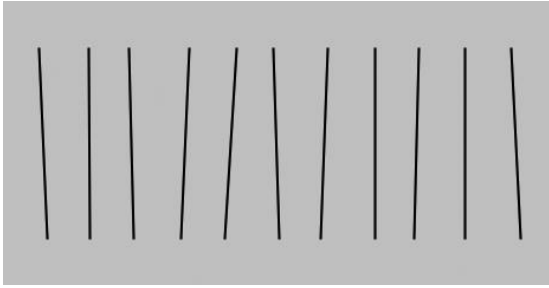
```

function draw() {
  var r = random(0, mouseX);
  print(r);
}

```

Example 8-8: Draw Randomly

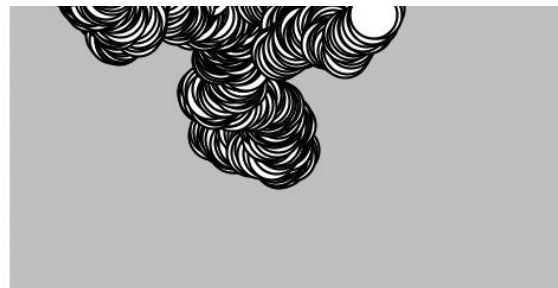
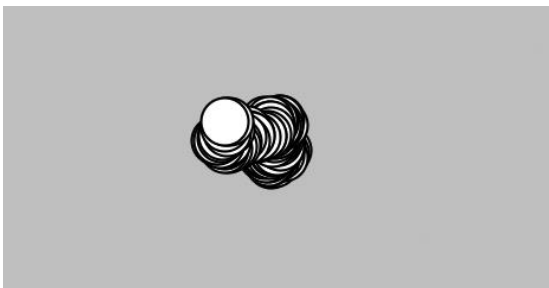
Building on [Example 8-7](#), this example uses the values from `random()` to change the position of lines on the canvas. When the mouse is at the left of the canvas, the change is small; as it moves to the right, the values from `random()` increase and the movement becomes more exaggerated. Because the `random()` function is inside the for loop, a new random value is calculated for each point of every line:



```
function setup() {  
  createCanvas(240, 120);  
}  
  
function draw() {  
  background(204);  
  for (var x = 20; x < width; x += 20) {  
    var mx = mouseX / 10;  
    var offsetA = random(-mx, mx);  
    var offsetB = random(-mx, mx);  
    line(x + offsetA, 20, x - offsetB, 100);  
  }  
}
```

Example 8-9: Move Shapes Randomly

When used to move shapes around on screen, random values can generate images that are more natural in appearance. In the following example, the position of the circle is modified by random values on each trip through draw(). Because the background() function is not used, past locations are traced:



```
var speed = 2.5;  
var diameter = 20;  
var x;  
var y;  
  
function setup() {  
  createCanvas(240, 120);  
  x = width/2;  
  y = height/2;  
  background(204);  
}
```

```
}
```

```
function draw() {  
  x += random(-speed, speed);  
  y += random(-speed, speed);  
  ellipse(x, y, diameter, diameter);  
}
```

If you watch this example long enough, you may see the circle leave the window and come back. This is left to chance, but we could add a few if structures or use the `constrain()` function to keep the circle from leaving the screen.

The `constrain()` function limits a value to a specific range, which can be used to keep `x` and `y` within the boundaries of the drawing canvas. By replacing the `draw()` in the preceding code with the following, you'll ensure that the ellipse will remain on the screen:

```
function draw() {  
  x += random(-speed, speed);  
  y += random(-speed, speed);  
  x = constrain(x, 0, width);  
  y = constrain(y, 0, height);  
  ellipse(x, y, diameter, diameter);  
}
```

NOTE

The `randomSeed()` function can be used to force `random()` to produce the same sequence of numbers each time a program is run. This is described further in the *p5.js Reference*.

Timers

Every p5.js program counts the amount of time that has passed since it was started. It counts in milliseconds (thousandths of a second), so after 1 second, the counter is at 1,000; after 5 seconds, it's at 5,000; after 1 minute, it's at 60,000. We can use this counter to trigger animations at specific times. The `millis()` function returns this counter value.

Example 8-10: Time Passes

You can watch the time pass when you run this program:

```
function draw() {  
  var timer = millis();  
  print(timer);  
}
```

Example 8-11: Triggering Timed Events

When paired with an if block, the values from `millis()` can be used to sequence animation and events within a program. For instance, after two seconds have elapsed, the code inside the if block can trigger a change. In this example, variables called `time1` and `time2` determine when to change the value of the `x` variable:

```
var time1 = 2000;
var time2 = 4000;
var x = 0;

function setup() {
  createCanvas(480, 120);
}

function draw() {
  var currentTime = millis();
  background(204);
  if (currentTime > time2) {
    x -= 0.5;
  } else if (currentTime > time1) {
    x += 2;
  }
  ellipse(x, 60, 90, 90);
}
```

Circular

If you're a trigonometry ace, you already know how amazing the *sine* and *cosine* functions are. If you're not, we hope the next examples will trigger your interest. We won't discuss the math in detail here, but we'll show a few applications to generate fluid motion.

[Figure 8-2](#) shows a visualization of sine wave values and how they relate to angles. At the top and bottom of the wave, notice how the rate of change (the change on the vertical axis) slows down, stops, then switches direction. It's this quality of the curve that generates interesting motion.

The `sin()` and `cos()` functions in `p5.js` return values between -1 and 1 for the sine or cosine of the specified angle. Like `arc()`, the angles must be given in radian values (see [Example 3-7](#) and [Example 3-8](#) for a reminder of how radians work). To be useful for drawing, the float values returned by `sin()` and `cos()` are usually multiplied by a larger value.

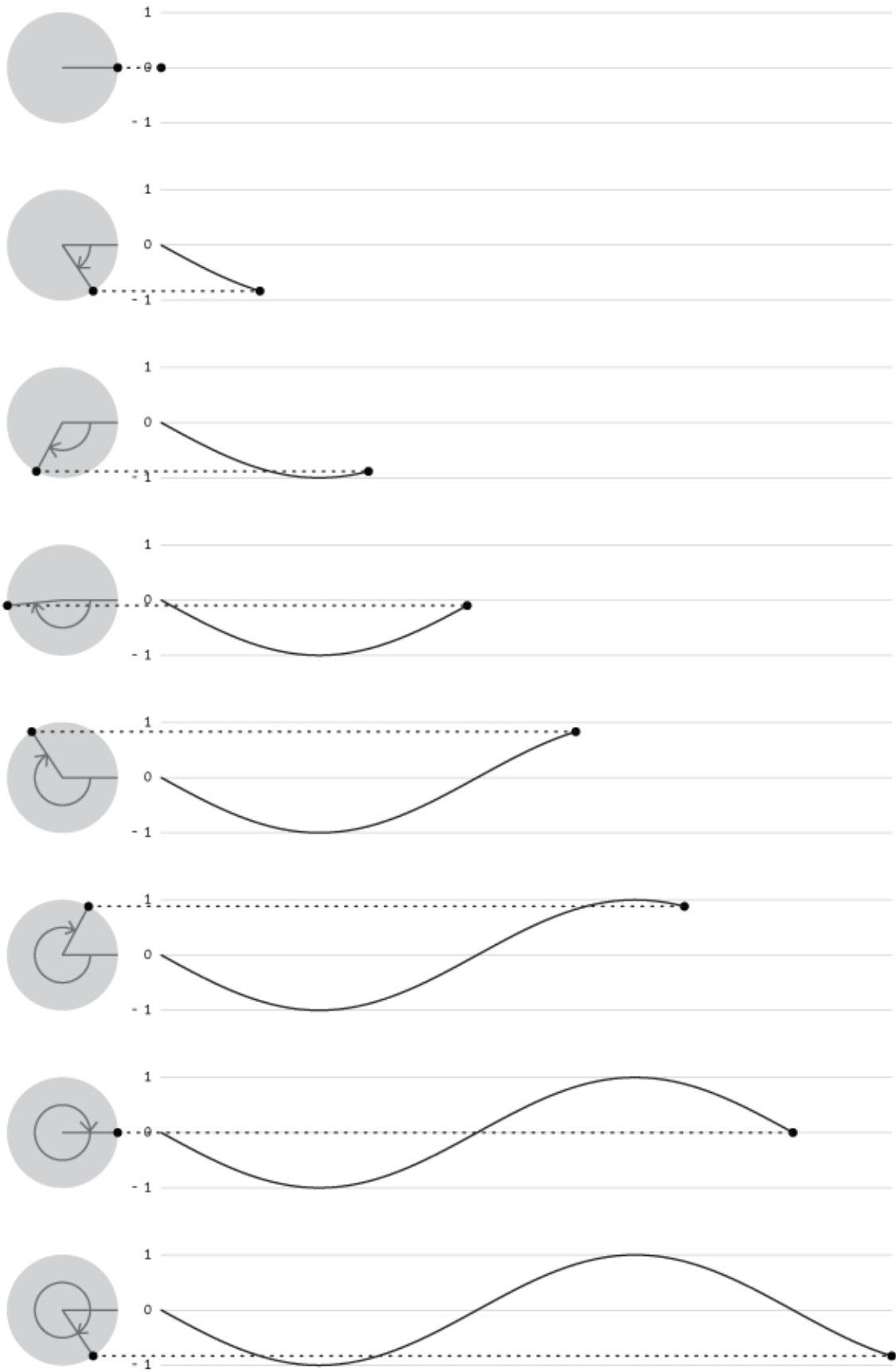


Figure 8-2. A sine wave is created by tracing the sine values of an angle that moves around a circle

Example 8-12: Sine Wave Values

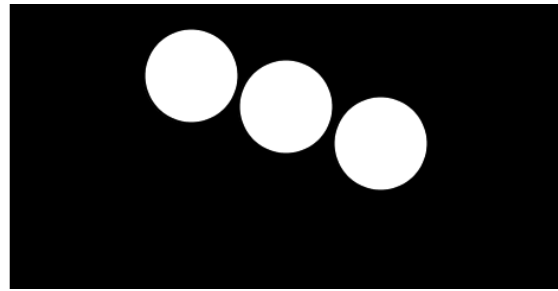
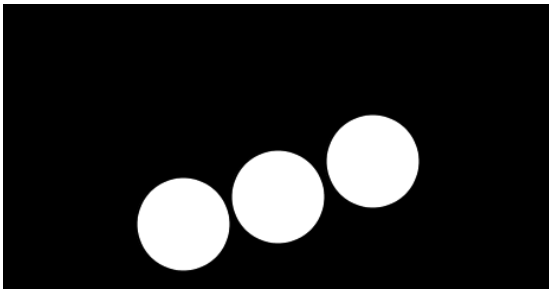
This example shows how values for `sin()` cycle from -1 to 1 as the angle increases. With the `map()` function, the `sinval` variable is converted from this range to values from 0 to 255 . This new value is used to set the background color of the canvas:

```
var angle = 0.0;

function draw() {
  var sinval = sin(angle);
  print(sinval);
  var gray = map(sinval, -1, 1, 0, 255);
  background(gray);
  angle += 0.1;
}
```

Example 8-13: Sine Wave Movement

This example shows how these values can be converted into movement:



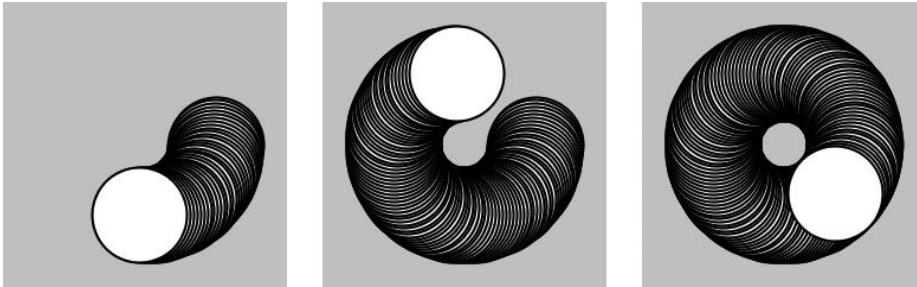
```
var angle = 0.0;
var offset = 60;
var scalar = 40;
var speed = 0.05;

function setup() {
  createCanvas(240, 120);
}

function draw() {
  background(0);
  var y1 = offset + sin(angle) * scalar;
  var y2 = offset + sin(angle + 0.4) * scalar;
  var y3 = offset + sin(angle + 0.8) * scalar;
  ellipse(80, y1, 40, 40);
  ellipse(120, y2, 40, 40);
  ellipse(160, y3, 40, 40);
  angle += speed;
}
```

Example 8-14: Circular Motion

When `sin()` and `cos()` are used together, they can produce circular motion. The `cos()` values provide the x coordinates, and the `sin()` values provide the y coordinates. Both are multiplied by a variable named `scalar` to change the radius of the movement and summed with an `offset` value to set the center of the circular motion:



```
var angle = 0.0;  
var offset = 60;  
var scalar = 30;  
var speed = 0.05;
```

```
function setup() {  
  createCanvas(120, 120);  
  background(204);  
}
```

```
function draw() {  
  var x = offset + cos(angle) * scalar;  
  var y = offset + sin(angle) * scalar;  
  ellipse(x, y, 40, 40);  
  angle += speed;  
}
```

Example 8-15: Spirals

A slight change made to increase the scalar value at each frame produces a spiral, rather than a circle:



```
var angle = 0.0;
```



```

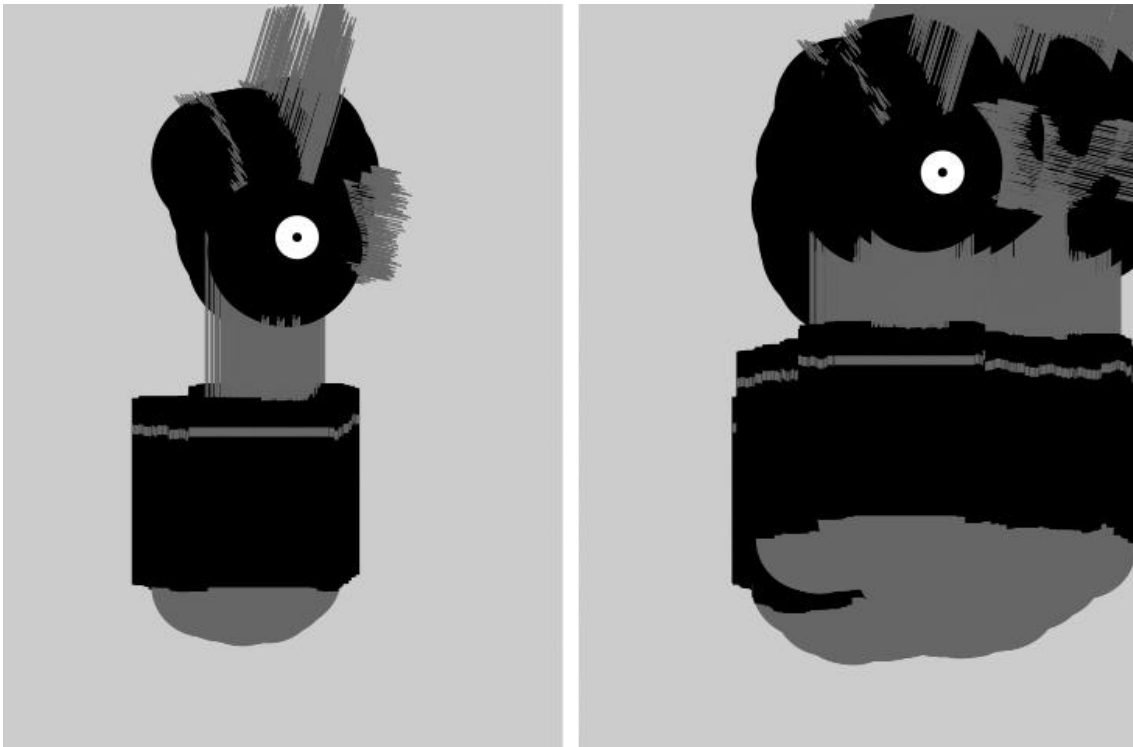
var offset = 60;
var scalar = 2;
var speed = 0.05;

function setup() {
  createCanvas(120, 120);
  fill(0);
  background(204);
}

function draw() {
  var x = offset + cos(angle) * scalar;
  var y = offset + sin(angle) * scalar;
  ellipse(x, y, 2, 2);
  angle += speed;
  scalar += speed;
}

```

Robot 6: Motion



In this example, the techniques for random and circular motion are applied to the robot. The background() was removed to make it easier to see how the robot's position and body change.

At each frame, a random number between -4 and 4 is added to the x coordinate, and a random number between -1 and 1 is added to the y coordinate. This causes the robot to move more from left to right than top to bottom. Numbers calculated from the $\sin()$ function change the height of the neck so it oscillates between 50 and 110 pixels high:

```

var x = 180;      // x coordinate
var y = 400;     // y coordinate
var bodyHeight = 153; // Body height
var neckHeight = 56; // Neck height
var radius = 45;  // Head radius
var angle = 0.0;  // Angle for motion

function setup() {
  createCanvas(360, 480);
  ellipseMode(RADIUS);
  background(204);
}

function draw() {
  // Change position by a small random amount
  x += random(-4, 4);
  y += random(-1, 1);

  // Change height of neck
  neckHeight = 80 + sin(angle) * 30;
  angle += 0.05;

  // Adjust the height of the head
  var ny = y - bodyHeight - neckHeight - radius;

  // Neck
  stroke(102);
  line(x+2, y-bodyHeight, x+2, ny);
  line(x+12, y-bodyHeight, x+12, ny);
  line(x+22, y-bodyHeight, x+22, ny);

  // Antennae
  line(x+12, ny, x-18, ny-43);
  line(x+12, ny, x+42, ny-99);
  line(x+12, ny, x+78, ny+15);

  // Body
  noStroke();
  fill(102);
  ellipse(x, y-33, 33, 33);
  fill(0);
  rect(x-45, y-bodyHeight, 90, bodyHeight-33);
  fill(102);
  rect(x-45, y-bodyHeight+17, 90, 6);

  // Head
  fill(0);
  ellipse(x+12, ny, radius, radius);
  fill(255);

```

```
ellipse(x+24, ny-6, 14, 14);  
fill(0);  
ellipse(x+24, ny-6, 3, 3);  
}
```

Chapter 9. Functions

Functions are the basic building blocks for p5.js programs. They have appeared in every example we've presented. For instance, we've frequently used the `createCanvas()` function, the `line()` function, and the `fill()` function. This chapter shows how to write new functions to extend the capabilities of p5.js beyond its built-in features.

The power of functions is modularity. Functions are independent software units that are used to build more complex programs—like LEGO bricks, where each type of brick serves a specific purpose, and making a complex model requires using the different parts together. As with functions, the true power of these bricks is the ability to build many different forms from the same set of elements. The same group of LEGOs that makes a spaceship can be reused to construct a truck, a skyscraper, and many other objects.

Functions are helpful if you want to draw a more complex shape like a tree over and over. The function to draw the tree shape would be made up of p5.js's built-in functions, like `line()`, that create the form. After the code to draw the tree is written, you don't need to think about the details of tree drawing again—you can simply write `tree()` (or whatever you named the function) to draw the shape. Functions allow a complex sequence of statements to be abstracted, so you can focus on the higher-level goal (such as drawing a tree), and not the details of the implementation (the `line()` functions that define the tree shape). Once a function is defined, the code inside the function need not be repeated again.

Function Basics

A computer runs a program one line at a time. When a function is run, the computer jumps to where the function is defined and runs the code there, then jumps back to where it left off.

Example 9-1: Roll the Dice

This behavior is illustrated with the `rollDice()` function written for this example. When a program starts, it runs the code in `setup()` and then stops. The program takes a detour and runs the code inside `rollDice()` each time it appears:

```
function setup() {  
  print("Ready to roll!");  
  rollDice(20);  
  rollDice(20);  
  rollDice(6);  
  print("Finished.");  
}
```

```

}

function rollDice(numSides) {
  var d = 1 + int(random(numSides));
  print("Rolling... " + d);
}

```

The two lines of code in `rollDice()` select a random number between 1 and the number of sides on the dice, and prints that number to the console. Because the numbers are random, you'll see different numbers each time the program is run:

```

Ready to roll!
Rolling... 20
Rolling... 11
Rolling... 1
Finished.

```

Each time the `rollDice()` function is run inside `setup()`, the code within the function runs from top to bottom, then the program continues on the next line within `setup()`.

The `random()` function (described in [“Random”](#)) returns a number from 0 up to (but not including) the number specified. So `random(6)` returns a number between 0 and 5.99999. . . Because `random()` returns a decimal point number, we also use `int()` to convert it to an integer. So `int(random(6))` will return 0, 1, 2, 3, 4, or 5. Then we add 1 so that the number returned is between 1 and 6 (like a die). Like many other cases in this book, counting from 0 makes it easier to use the results of `random()` with other calculations.

Example 9-2: Another Way to Roll

If an equivalent program were written without the `rollDice()` function, it might look like this:

```

function setup() {
  print("Ready to roll!");
  var d1 = 1 + int(random(20));
  print("Rolling... " + d1);
  var d2 = 1 + int(random(20));
  print("Rolling... " + d2);
  var d3 = 1 + int(random(6));
  print("Rolling... " + d3);
  print("Finished.");
}

```

The `rollDice()` function in [Example 9-1](#) makes the code easier to read and maintain. The program is clearer, because the name of the function clearly states its purpose. In this example, we see the `random()` function in `setup()`, but its use is not as obvious. The number of sides on the die is also clearer with a function: when the code

says `rollDice(6)`, it's obvious that it's simulating the roll of a six-sided die. Also, [Example 9-1](#) is easier to maintain, because information is not repeated. The phrase `Rolling...` is repeated three times here. If you want to change that text to something else, you would need to update the program in three places, rather than making a single edit inside the `rollDice()` function. In addition, as you'll see in [Example 9-5](#), a function can also make a program much shorter (and therefore easier to maintain and read), which helps reduce the potential number of bugs.

Make a Function

In this section, we'll draw an owl to explain the steps involved in making a function.

Example 9-3: Draw the Owl

First, we'll draw the owl without using a function:



```
function setup() {  
  createCanvas(480, 120);  
}  
  
function draw() {  
  background(204);  
  translate(110, 110);  
  stroke(0);  
  strokeWeight(70);  
  line(0, -35, 0, -65); // Body  
  noStroke();  
  fill(204);  
  ellipse(-17.5, -65, 35, 35); // Left eye dome  
  ellipse(17.5, -65, 35, 35); // Right eye dome  
  arc(0, -65, 70, 70, 0, PI); // Chin  
  fill(0);  
  ellipse(-14, -65, 8, 8); // Left eye  
  ellipse(14, -65, 8, 8); // Right eye  
  quad(0, -58, 4, -51, 0, -44, -4, -51); // Beak  
}
```

Notice that `translate()` is used to move the origin (0,0) to 110 pixels over and 110 pixels down. Then the owl is drawn relative to (0,0), with its coordinates sometimes positive and negative as it's centered around the new 0,0 point. (See [Figure 9-1](#).)

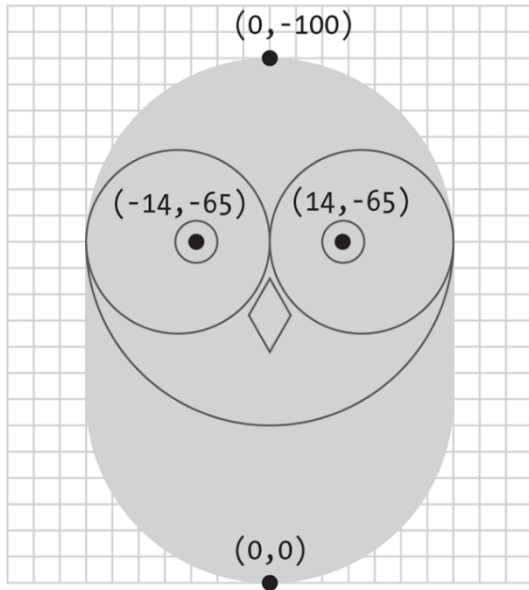
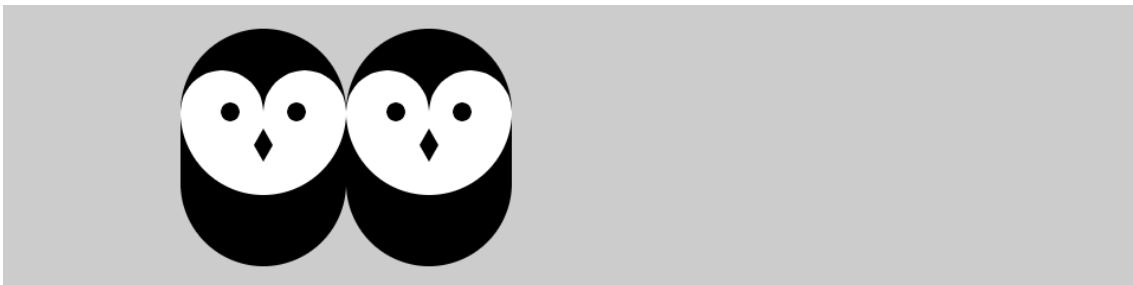


Figure 9-1. The owl's coordinates

Example 9-4: Two's Company

The code presented in [Example 9-3](#) is reasonable if there is only one owl, but when we draw a second, the length of the code is nearly doubled:



```
function setup() {
  createCanvas(480, 120);
}

function draw() {
  background(204);

  // Left owl
  translate(110, 110);
  stroke(0);
  strokeWeight(70);
  line(0, -35, 0, -65); // Body
  noStroke();
  fill(204);
  ellipse(-17.5, -65, 35, 35); // Left eye dome
```

```

ellipse(17.5, -65, 35, 35); // Right eye dome
arc(0, -65, 70, 70, 0, PI); // Chin
fill(0);
ellipse(-14, -65, 8, 8); // Left eye
ellipse(14, -65, 8, 8); // Right eye
quad(0, -58, 4, -51, 0, -44, -4, -51); // Beak

// Right owl
translate(70, 0);
stroke(0);
strokeWeight(70);
line(0, -35, 0, -65); // Body
noStroke();
fill(255);
ellipse(-17.5, -65, 35, 35); // Left eye dome
ellipse(17.5, -65, 35, 35); // Right eye dome
arc(0, -65, 70, 70, 0, PI); // Chin
fill(0);
ellipse(-14, -65, 8, 8); // Left eye
ellipse(14, -65, 8, 8); // Right eye
quad(0, -58, 4, -51, 0, -44, -4, -51); // Beak
}

```

The program grew from 21 lines to 34: the code to draw the first owl was cut and pasted into the program and a `translate()` was inserted to move it 70 pixels to the right. This is a tedious and inefficient way to draw a second owl, not to mention the headache of adding a third owl with this method. But duplicating the code is unnecessary, because this is the type of situation where a function can come to the rescue.

Example 9-5: An Owl Function

In this example, a function is introduced to draw two owls with the same code. If we make the code that draws the owl to the screen into a new function, the code need only appear once in the program:



```

function setup() {
  createCanvas(480, 120);
}

function draw() {
  background(204);

```

```

owl(110, 110);
owl(180, 110);
}

function owl(x, y) {
  push();
  translate(x, y);
  stroke(0);
  strokeWeight(70);
  line(0, -35, 0, -65); // Body
  noStroke();
  fill(255);
  ellipse(-17.5, -65, 35, 35); // Left eye dome
  ellipse(17.5, -65, 35, 35); // Right eye dome
  arc(0, -65, 70, 70, 0, PI); // Chin
  fill(0);
  ellipse(-14, -65, 8, 8); // Left eye
  ellipse(14, -65, 8, 8); // Right eye
  quad(0, -58, 4, -51, 0, -44, -4, -51); // Beak
  pop();
}

```

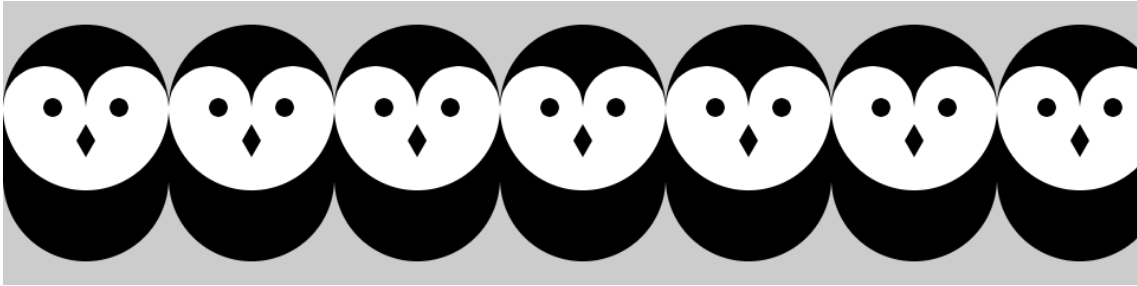
You can see from the illustrations that this example and [Example 9-4](#) have the same result, but this example is shorter, because the code to draw the owl appears only once, inside the aptly named `owl()` function. This code runs twice, because it's called twice inside `draw()`. The owl is drawn in two different locations because of the parameters passed into the function that set the x and y coordinates.

Parameters are an important part of functions, because they provide flexibility. We saw another example in the `rollDice()` function; the single parameter named `numSides` made it possible to simulate a 6-sided die, a 20-sided die, or a die with any number of sides. This is just like many other p5.js functions. For instance, the parameters to the `line()` function make it possible to draw a line from any pixel on the canvas to any other pixel. Without the parameters, the function would be able to draw a line only from one fixed point to another.

Each parameter is a variable that's created each time the function runs. When this example is run, the first time the owl function is called, the value of the x parameter is 110, and y is also 110. In the second use of the function, the value of x is 180 and y is again 110. Each value is passed into the function and then wherever the variable name appears within the function, it's replaced with the incoming value.

Example 9-6: Increasing the Surplus Population

Now that we have a basic function to draw the owl at any location, we can draw many owls efficiently by placing the function within a for loop and changing the first parameter each time through the loop:



```
function setup() {  
  createCanvas(480, 120);  
}  
  
function draw() {  
  background(204);  
  for (var x = 35; x < width + 70; x += 70) {  
    owl(x, 110);  
  }  
}  
  
// Insert owl() function from Example 9-5
```

It's possible to keep adding more and more parameters to the function to change different aspects of how the owl is drawn. Values could be passed in to change the owl's color, rotation, scale, or the diameter of its eyes.

Example 9-7: Owls of Different Sizes

In this example, we've added two parameters to change the gray value and size of each owl:



```
function setup() {  
  createCanvas(480, 120);  
}  
  
function draw() {  
  background(204);  
  randomSeed(0);  
  for (var i = 35; i < width + 40; i += 40) {  
    var gray = int(random(0, 102));
```

```

    var scalar = random(0.25, 1.0);
    owl(i, 110, gray, scalar);
  }
}

function owl(x, y, g, s) {
  push();
  translate(x, y);
  scale(s); // Set the scale
  stroke(g); // Set the gray value
  strokeWeight(70);
  line(0, -35, 0, -65); // Body
  noStroke();
  fill(255-g);
  ellipse(-17.5, -65, 35, 35); // Left eye dome
  ellipse(17.5, -65, 35, 35); // Right eye dome
  arc(0, -65, 70, 70, 0, PI); // Chin
  fill(g);
  ellipse(-14, -65, 8, 8); // Left eye
  ellipse(14, -65, 8, 8); // Right eye
  quad(0, -58, 4, -51, 0, -44, -4, -51); // Beak
  pop();
}

```

Return Values

Functions can make a calculation and then return a value to the main program. We've already used functions of this type, including `random()` and `sin()`. Notice that when this type of function appears, the return value is usually assigned to a variable:

```
var r = random(1, 10);
```

In this case, `random()` returns a value between 1 and 10, which is then assigned to the `r` variable.

A function that returns a value is also frequently used as a parameter to another function. For instance:

```
point(random(width), random(height));
```

In this case, the values from `random()` aren't assigned to a variable—they are passed as parameters to `point()` and used to position the point within the canvas.

Example 9-8: Return a Value

To make a function that returns a value, specify the data to be passed back with the keyword `return`. For instance, this example includes a function

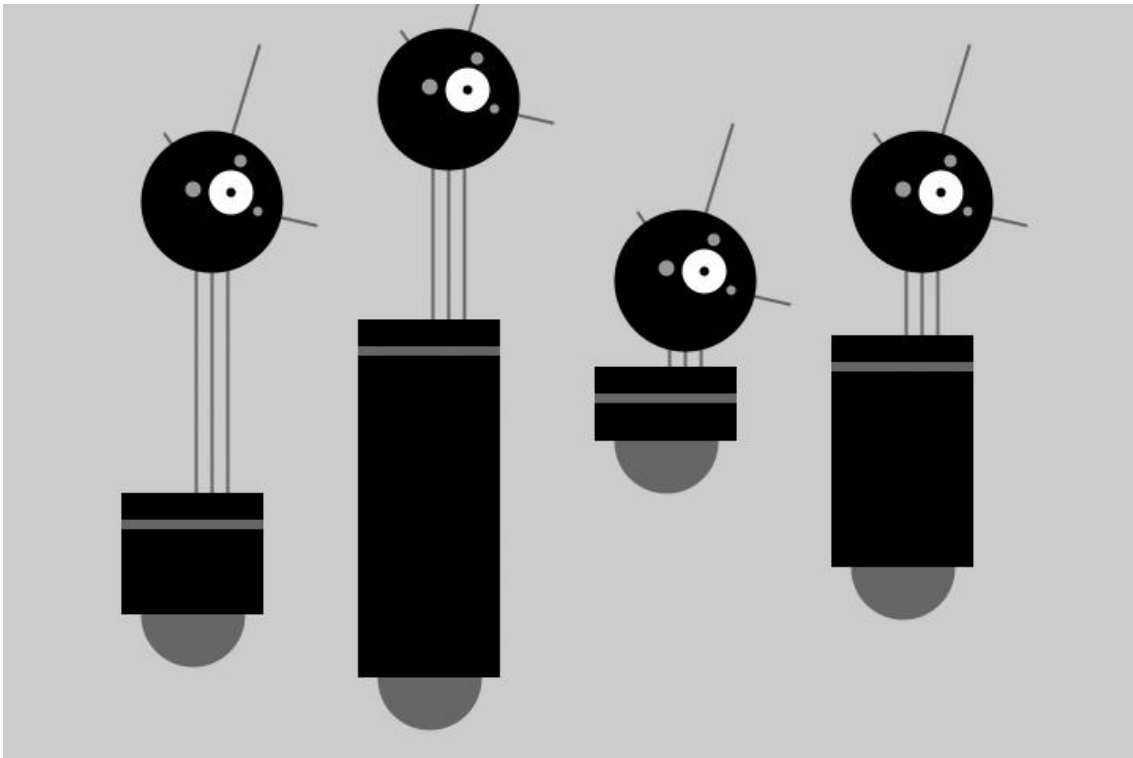
called `calculateMars()` that calculates the weight of a person or object on our neighboring planet:

```
function setup() {  
  var yourWeight = 132;  
  var marsWeight = calculateMars(yourWeight);  
  print(marsWeight);  
}
```

```
function calculateMars(w) {  
  var newWeight = w * 0.38;  
  return newWeight;  
}
```

Notice the last line of the block, which returns the variable `newWeight`. In the second line of `setup()`, that value is assigned to the variable `marsWeight`. (To see your own weight on Mars, change the value of the `yourWeight` variable to your weight.)

Robot 7: Functions



In contrast to Robot 2 (see [“Robot 2: Variables”](#)), this example uses a function to draw four robot variations within the same program. Because the `drawRobot()` function appears four times within `draw()`, the code within the `drawRobot()` block is run four times, each time with a different set of parameters to change the position and height of the robot’s body.

Notice how what were global variables in Robot 2 have now been isolated within the `drawRobot()` function. Because these variables apply only to drawing the robot, they

belong inside the curly braces that define the drawRobot() function block. Because the value of the radius variable doesn't change, it need not be a parameter. Instead, it is defined at the beginning of drawRobot():

```
function setup() {  
  createCanvas(720, 480);  
  strokeWeight(2);  
  ellipseMode(RADIUS);  
}
```

```
function draw() {  
  background(204);  
  drawRobot(120, 420, 110, 140);  
  drawRobot(270, 460, 260, 95);  
  drawRobot(420, 310, 80, 10);  
  drawRobot(570, 390, 180, 40);  
}
```

```
function drawRobot(x, y, bodyHeight, neckHeight) {
```

```
  var radius = 45;  
  var ny = y - bodyHeight - neckHeight - radius;
```

```
  // Neck  
  stroke(102);  
  line(x+2, y-bodyHeight, x+2, ny);  
  line(x+12, y-bodyHeight, x+12, ny);  
  line(x+22, y-bodyHeight, x+22, ny);
```

```
  // Antennae  
  line(x+12, ny, x-18, ny-43);  
  line(x+12, ny, x+42, ny-99);  
  line(x+12, ny, x+78, ny+15);
```

```
  // Body  
  noStroke();  
  fill(102);  
  ellipse(x, y-33, 33, 33);  
  fill(0);  
  rect(x-45, y-bodyHeight, 90, bodyHeight-33);  
  fill(102);  
  rect(x-45, y-bodyHeight+17, 90, 6);
```

```
  // Head  
  fill(0);  
  ellipse(x+12, ny, radius, radius);  
  fill(255);  
  ellipse(x+24, ny-6, 14, 14);  
  fill(0);
```

```
ellipse(x+24, ny-6, 3, 3);
fill(153);
ellipse(x, ny-8, 5, 5);
ellipse(x+30, ny-26, 4, 4);
ellipse(x+41, ny+6, 3, 3);
}
```

Chapter 10. Objects

Object-oriented programming (OOP) is a different way to think about your programs. *Objects* are also a way to group variables with related functions. Because you already know how to work with variables and functions, objects simply combine what you've already learned into a more understandable package.

Objects are important, because they break up ideas into smaller building blocks. This mirrors the natural world where, for instance, organs are made of tissue, tissue is made of cells, and so on. Similarly, as your code becomes more complicated, you must think in terms of smaller structures that form more complicated ones. It's easier to write and maintain smaller, understandable pieces of code that work together than it is to write one large piece of code that does everything at once.

Properties and Methods

A software object is a collection of related variables and functions. In the context of objects, a variable is called a *property* (or *instance variable*) and a function is called a *method*. Properties and methods work just like the variables and functions covered in earlier chapters, but we'll use the new terms to emphasize that they are a part of an object. To say it another way, an object combines related data (properties) with related actions and behaviors (methods). The idea is to group together related data with related methods that act on that data.

For instance, to model a radio, think about what parameters can be adjusted and the actions that can affect those parameters:

Properties

volume, frequency, band(FM, AM), power(on, off)

Methods

setVolume, setFrequency, setBand

Modeling a simple mechanical device is easy compared to modeling an organism like an ant or a person. It's not possible to reduce such complex organisms to a few properties and methods, but it is possible to model enough to create an interesting simulation. *The Sims* video game is a clear example. This game is played by managing the daily activities of simulated people. The characters have enough personality to make a playable, addictive game, but no more. In fact, they have only five personality attributes: neat, outgoing, active, playful, and nice. With the knowledge that it's possible to make a highly simplified model of complex organisms, we could start programming an ant with only a few properties and methods:

Properties

type(worker, soldier), weight, length
Methods
walk, pinch, releasePheromones, eat

If you made a list of an ant's properties and methods, you might choose to focus on different aspects of the ant to model. There's no right way to make a model, as long as you make it appropriate for the purpose of your program's goals.

Define a Constructor

To create an object, start by defining a constructor function. A *constructor function* is the specification for an object. Using an architectural analogy, a constructor function is like a blueprint for a house, and the object is like the house itself. Each house made from the blueprint can have variations, and the blueprint is only the specification, not a built structure. For example, one house can be blue and the other red; one house might come with a fireplace and the other without. Likewise with objects, the constructor defines the data types and behaviors, but each object (house) made from a single constructor function (blueprint) has variables (color, fireplace) that are set to different values. To use a more technical term, each object is an *instance* and each instance has its own set of properties and methods.

Before you write a constructor function, we recommend a little planning. Think about what properties and methods your objects should have. Do a little brainstorming to imagine all the possible options and then prioritize and make your best guess about what will work. You'll make changes during the programming process, but it's important to have a good start.

For your properties, select clear names. The properties of an object can hold any type of data. An object can simultaneously hold many booleans, numbers, images, strings, and so on. Keep in mind that one reason to make an object is to group together related data elements. For your methods, select clear names and decide the return values (if any). The methods are used to change the values of the properties and to perform actions based on the properties' values.

For our first constructor function, we'll convert [Example 8-9](#) from earlier in the book. We start by making a list of the properties from the example:

```
var x  
var y  
var diameter  
var speed
```

The next step is to figure out what methods might be useful for the object. In looking at the draw() function from the example we're adapting, we see two primary components. The position of the shape is updated and drawn to the screen. Let's create two methods for our object, one for each task:

```
function move()
```

`function display()`

Neither of these methods return a value. Once we've determined the properties and methods the object should have, we'll write our constructor function to assign them to each instance of the object we create ([Figure 10-1](#)).

```
var red, blue;

function setup() {
  createCanvas(400, 400);
  red = new Train("Red Line", 90);
  blue = new Train("Blue Line", 120);
}

function Train (tempName, tempDistance) {
  this.name = tempName;
  this.distance = tempDistance;
}
}
```

Assign "Red Line" to the name variable for the red object

Assign 90 to the distance variable for the red object

```
var red, blue;

function setup() {
  createCanvas(400, 400);
  red = new Train("Red Line", 90);
  blue = new Train("Blue Line", 120);
}

function Train (tempName, tempDistance) {
  this.name = tempName;
  this.distance = tempDistance;
}
}
```

Assign "Blue Line" to the name variable for the blue object

Assign 120 to the distance variable for the blue object

Figure 10-1. Passing values into the constructor to set the values for an object's properties

The code inside the constructor function is run once when the object is first created. To create the constructor function, we'll follow three steps:

1. Create a function block.
2. Add the properties and assign values to them.
3. Add the methods.

First, we create a function block for our constructor:

```
function JitterBug() {  
  
}
```

Notice that the name `JitterBug` is uppercase. Naming the constructor function with an uppercase letter isn't required, but it is a convention (that we strongly encourage) used to denote that it's a constructor. (The keyword `function`, however, must be lowercase because it's a rule of the programming language.)

Second, we add the properties. JavaScript has a special keyword, `this`, that you can use within the constructor function to refer to the current object. When declaring a property of an object, we leave off the symbol `var`, and instead prepend the variable name with `this.` to indicate that we are assigning a property, a variable of the object. We could declare and assign the `speed` property as follows:

```
function JitterBug() {  
  this.speed = 0.5;  
}
```

While we are doing this, we have to decide which properties will have their values passed in through the *constructor*. As a rule of thumb, property values that you want to be different for each instance are passed in through the constructor, and the other property values can be defined when they are declared within the constructor, as `speed` is in this case. For the `JitterBug` object, we've decided that the values for `x`, `y`, and `diameter` will be passed in. Each of the values passed in is assigned to a temporary variable that exists only while the code inside the constructor is run. To clarify this, we've added the name `temp` to each of these variables, but they can be named with any terms that you prefer. They are used only to assign the values to the properties that are a part of the object. So we add `tempX`, `tempY`, and `tempDiameter` as parameters for the function, and the properties are declared and assigned as follows:

```
function JitterBug(tempX, tempY, tempDiameter) {  
  this.x = tempX;  
  this.y = tempY;  
  this.diameter = tempDiameter;  
  this.speed = 0.5; // Same for every instance  
}
```


The last step is to add the methods. This is just like writing functions, but here they are contained within the constructor function, and the first line is written a bit differently. Normally, a function to update variables might be written like this:

```
function move() {  
  x += random(-speed, speed);  
  y += random(-speed, speed);  
}
```

Because we want to make this function a method of the object, we again need to use the `this` keyword. The preceding function is converted into a method like this:

```
this.move = function() {  
  this.x += random(-this.speed, this.speed);  
  this.y += random(-this.speed, this.speed);  
};
```

The first line looks a little strange, but the way to interpret it is “create an instance variable (property) called `move`, and assign its value to be this function.” Then, any time we refer to properties of the object, we again use `this.`, just as we do when they’re initially declared. Putting it together in the constructor looks like this:

```
function JitterBug(tempX, tempY, tempDiameter) {  
  
  this.x = tempX;  
  this.y = tempY;  
  this.diameter = tempDiameter;  
  this.speed = 2.5;  
  
  this.move = function() {  
    this.x += random(-this.speed, this.speed);  
    this.y += random(-this.speed, this.speed);  
  };  
  
  this.display = function() {  
    ellipse(this.x, this.y, this.diameter, this.diameter);  
  };  
  
}
```

Also note the code spacing. Every line within the constructor is indented a few spaces to show that it’s inside the block. Within the methods, the code is spaced again to clearly show the hierarchy.

Create Objects

Now that you have defined a constructor function, to use it in a program you must create an object instance from that constructor. There are two steps to create an object:

1. Declare the object variable.
2. Create (initialize) the object with the keyword `new`.

Example 10-1: Make an Object

To make your first object, we'll start by showing how this works within a p5.js sketch and then continue by explaining each part in depth:



```
var bug;

function setup() {
  createCanvas(480, 120);
  background(204);
  // Create object and pass in parameters
  bug = new JitterBug(width/2, height/2, 20);
}

function draw() {
  bug.move();
  bug.display();
}

// Put a copy of the JitterBug constructor here
```

We declare object variables in the same way as all other variables—the object is declared by writing the keyword `var` followed by a name for the variable:

```
var bug;
```

The second step is to initialize the object with the keyword `new`. It makes space for the object in memory with all its properties and methods. The name of the constructor is written to the right of the `new` keyword, followed by the parameters into the constructor, if any:

```
bug = new JitterBug(width/2, height/2, 20);
```

The three numbers within the parentheses are the parameters that are passed into the JitterBug constructor function. The number and order of these parameters must match those of the constructor.

Example 10-2: Make Multiple Objects

In [Example 10-1](#), we see something else new: the period (dot) that's used to access the object's methods inside of draw(). The dot operator is used to join the name of the object with its properties and methods. It mirrors the way we use this. inside the constructor function, but when we refer to it outside the constructor, this is replaced by the variable name.

This becomes clearer in this example, where two objects are made from the same constructor. The jit.move() function refers to the move() method that belongs to the object named jit, and bug.move() refers to the move() method that belongs to the object named bug:



```
var jit;  
var bug;  
  
function setup() {  
  createCanvas(480, 120);  
  background(204);  
  jit = new JitterBug(width * 0.33, height/2, 50);  
  bug = new JitterBug(width * 0.66, height/2, 10);  
}  
  
function draw() {  
  jit.move();  
  jit.display();  
  bug.move();  
  bug.display();  
}  
  
// Put a copy of the JitterBug constructor here
```

Now that the constructor exists as its own module of code, any changes will modify the objects made from it. For instance, you could add a property to the JitterBug constructor that controls the color, or another that determines its size. These values can be passed in using the constructor or assigned using additional methods, such

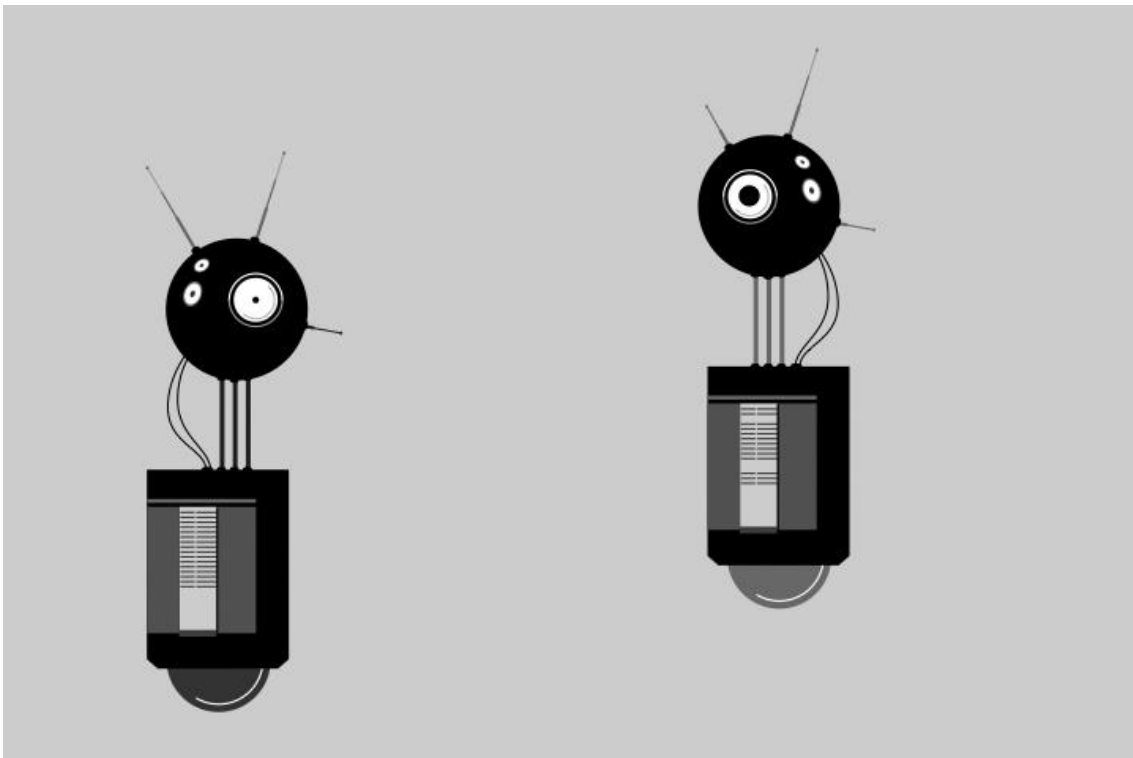
as `setColor()` or `setSize()`. And because it's a self-contained unit, you can also use the `JitterBug` constructor in another sketch.

Now is a good time to learn about using multiple files in JavaScript. Spreading your code across more than one file makes longer code easier to edit and more manageable in general. A new file is usually created for each constructor, which reinforces the modularity of working with objects and makes the code easy to find.

Create a new file in the same folder as your current `sketch.js` file. You can name it anything you like, but it is a good idea to name it `JitterBug.js` for organization. Move the `JitterBug` constructor function into this new file. Link the `JitterBug.js` file into your HTML file by adding a line in the HEAD below the line where you link in the `sketch.js` file:

```
<script type="text/javascript" src="sketch.js"></script>  
<script type="text/javascript" src="JitterBug.js"></script>
```

Robot 8: Objects



A software object combines methods (functions) and properties (variables) into one unit. The `Robot` constructor function in this example defines all of the robot objects that will be created from it. Each `Robot` object has its own set of properties to store a position and the illustration that will draw to the screen. Each has methods to update the position and display the illustration.

The parameters for `bot1` and `bot2` in `setup()` define the x and y coordinates and the `.svg` file that will be used to depict the robot. The `tempX` and `tempY` parameters are passed into the constructor and assigned to the `xpos` and `ypos` properties.

The `imgPath` parameter is used to load the related illustration. The objects (`bot1` and `bot2`) draw at their own location and with a different illustration because they each have unique values passed into the objects through their constructors:

```
var img1;
var img2;

var bot1;
var bot2;

function preload() {
  img1 = loadImage("robot1.svg");
  img2 = loadImage("robot2.svg");
}

function setup() {
  createCanvas(720, 480);
  bot1 = new Robot(img1, 90, 80);
  bot2 = new Robot(img2, 440, 30);
}

function draw() {
  background(204);

  // Update and display first robot
  bot1.update();
  bot1.display();

  // Update and display second robot
  bot2.update();
  bot2.display();
}

function Robot(img, tempX, tempY) {
  // Set initial values for properties
  this.xpos = tempX;
  this.ypos = tempY;
  this.angle = random(0, TWO_PI);
  this.botImage = img;
  this.yoffset = 0.0;

  // Update the properties
  this.update = function() {
    this.angle += 0.05;
    this.yoffset = sin(this.angle) * 20;
  }

  // Draw the robot to the screen
  this.display = function() {
```

```
    image(this.botImage, this.xpos, this.ypos + this.yoffset);  
  }  
}
```

<https://github.com/lmccart/gswp5.js-code>.