1/Hello

p5.js is for writing software to make images, animations, and interactions. The idea is to write a single line of code, and have a circle show up on the screen. Add a few more lines of code, and the circle follows the mouse. Another line of code, and the circle changes color when the mouse is pressed. We call this *sketching* with code. You write one line, then add another, then another, and so on. The result is a program created one piece at a time.

Programming courses typically focus on structure and theory first. Anything visual—an interface, an animation—is considered a dessert to be enjoyed only after finishing your vegetables, usually several weeks of studying algorithms and methods. Over the years, we've watched many friends try to take such courses only to drop out after the first lecture or after a long, frustrating night before the first assignment deadline. What initial curiosity they had about making the computer work for them was lost because they couldn't see a path from what they had to learn first to what they wanted to create.

p5.js offers a way to learn programming through creating interactive graphics. There are many possible ways to teach coding, but students often find encouragement and motivation in immediate visual feedback. p5.js provides this feedback, and its emphasis on images, sketching, and community is discussed in the next few pages.

Sketching and Prototyping

Sketching is a way of thinking; it's playful and quick. The basic goal is to explore many ideas in a short amount of time. In our own work, we usually start by sketching on paper and then moving the results into code. Ideas for animation and interactions are usually sketched as storyboards with notations. After making some software sketches, the best ideas are selected and combined into prototypes (Figure 1-1). It's a cyclical process of making, testing, and improving that moves back and forth between paper and screen.

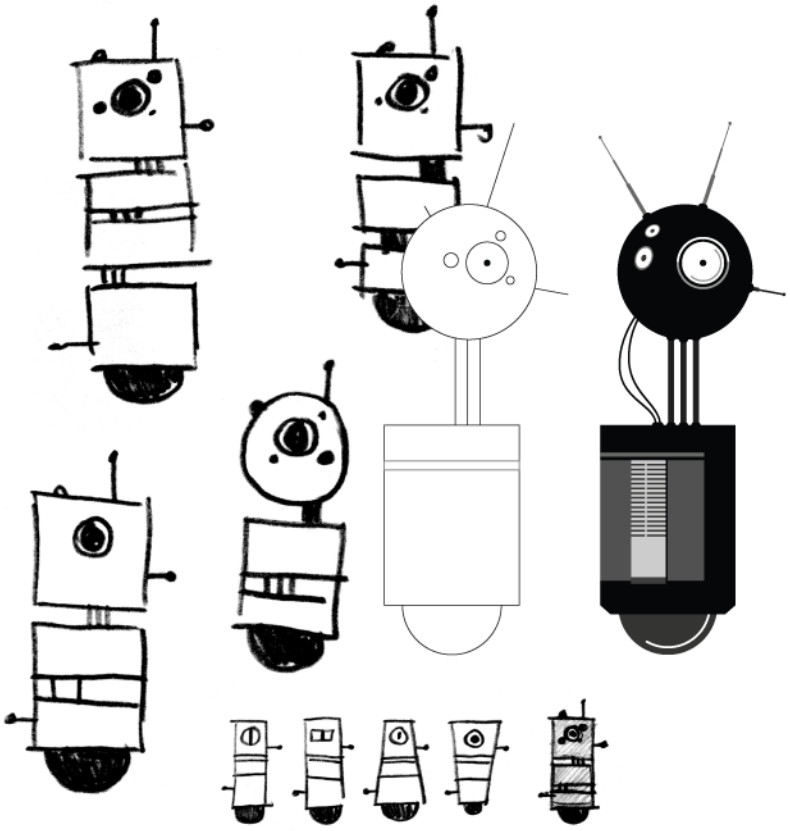


Figure 1-1. *As drawings move from sketchbook to screen, new possibilities emerge*

Flexibility

Like a software utility belt, p5.js consists of many tools that work together in different combinations. As a result, it can be used for quick hacks or for in-depth research. Because a p5.js program can be as short as a few lines or as long as thousands, there's room for growth and variation. Libraries extend p5.js even further into domains including playing with sound and adding buttons, sliders, input boxes, and webcam capture with HTML.

Giants

People have been making pictures with computers since the 1960s, and there's much to be learned from this history. For example, before computers could display to CRT or LCD screens, huge plotter machines (Figure 1-2) were used to draw images. In life, we all stand on the shoulders of giants, and the titans for p5.js include thinkers from design, computer graphics, art, architecture, statistics, and the spaces between. Have a look at Ivan Sutherland's *Sketchpad* (1963), Alan Kay's *Dynabook* (1968), and the many artists featured in Ruth Leavitt's *Artist and Computer*[1] (Harmony Books, 1976). The ACM SIGGRAPH and Ars Electronica archives provide fascinating glimpses into the history of graphics and software.

1 *http://www.atariarchives.org/artist/*

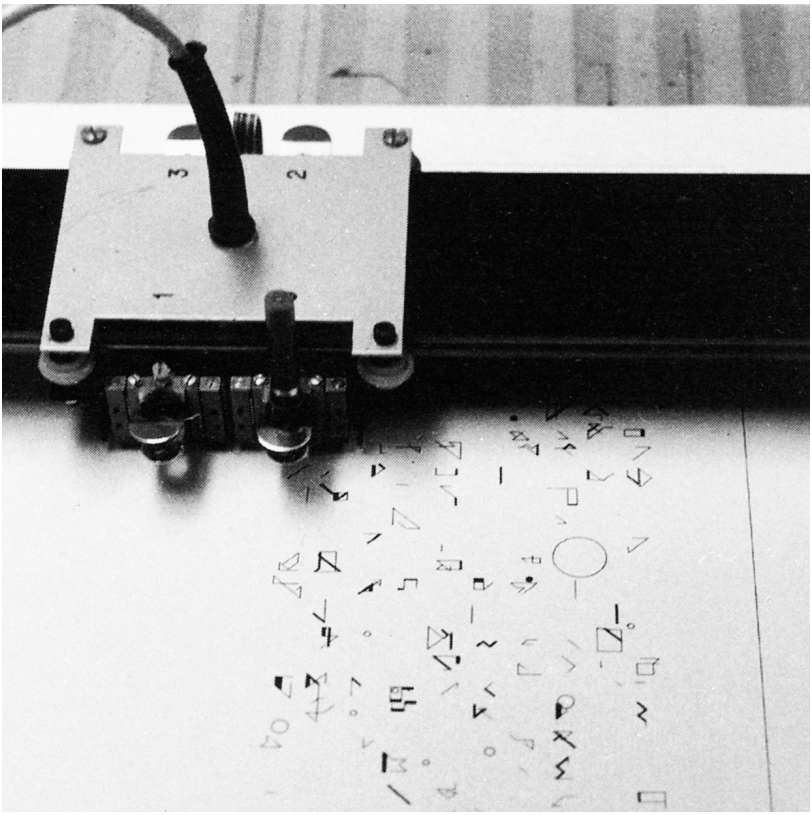


Figure 1-2. *Drawing demonstration by Manfred Mohr at Musée d'Art Moderne de la Ville de Paris using the Benson plotter and a digital computer on May 11, 1971 (photo by Rainer Mürle, courtesy bitforms gallery, New York)*

Family Tree

Like human languages, programming languages belong to families of related languages. p5.js is a dialect of a programming language called JavaScript. The language syntax is almost identical, but p5.js adds custom features related to graphics and interaction (Figure 1-3) and provides easier access to native HTML5 features already supported by the browser. Because of these shared features, learning p5.js is an entry-level step to programming in other languages and using different software tools.

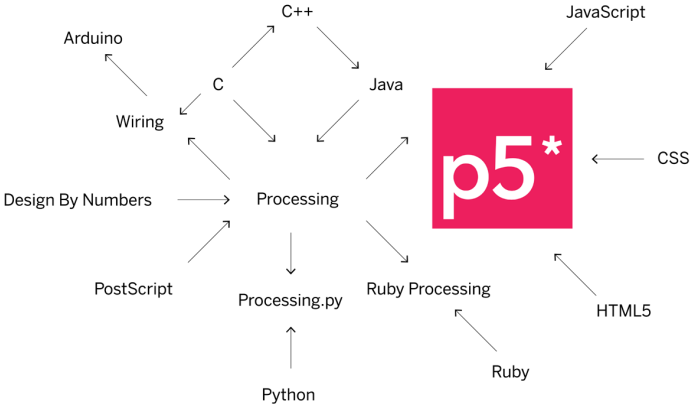


Figure 1-3. *p5.js has a large family of related languages and programming environments*

Join In

Thousands of people use p5.js every day. Like them, you can download p5.js for free. You even have the option to modify the p5.js code to suit your needs. p5.js is a *FLOSS* project (that is, *free/libre/open source software*), and in the spirit of community, we encourage you to participate by sharing your projects and knowledge online at *http://p5js.org* (Figure 1-4).

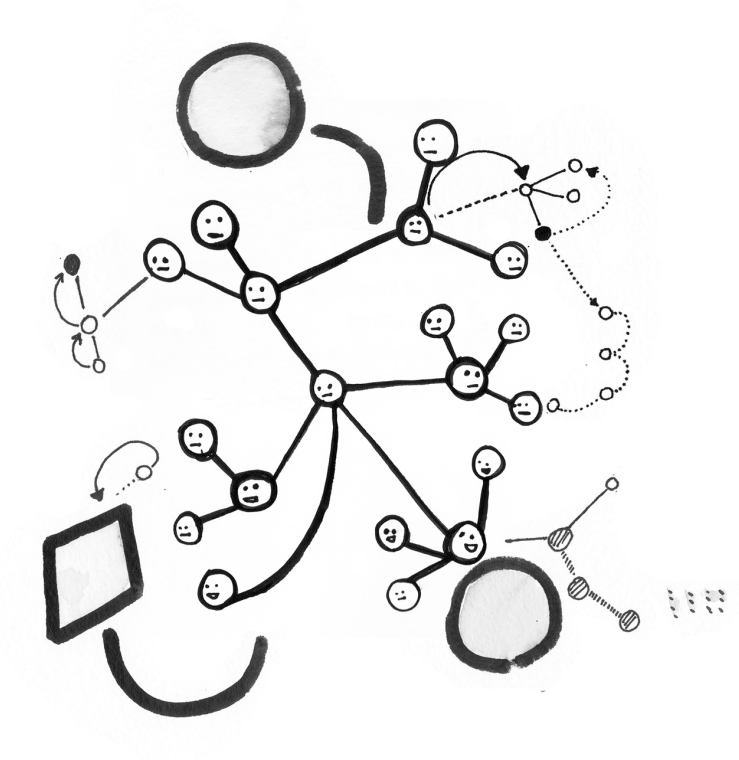


Figure 1-4. *p5.js is fueled by a community of people contributing through the Internet*

```
function setup()
{
  createCanvas(600, 400);
  line(15, 25, 70, 90);
}
```

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" on page 10.

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

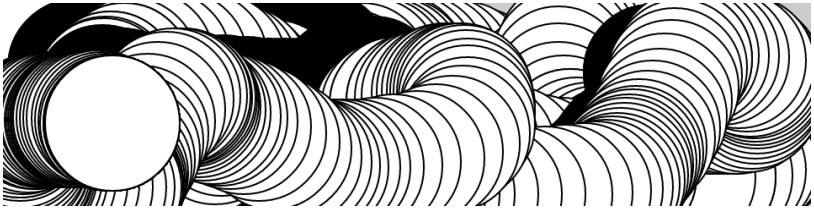
example code exactly. The numbers should be contained within parentheses and have commas between each of them. The line should end with a semicolon.

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.

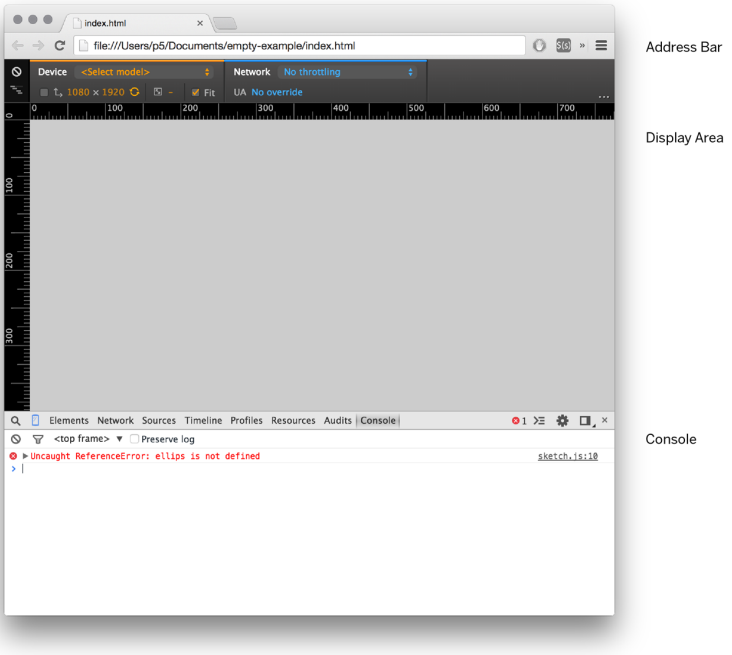


Figure 2-1. *Example view of an error in the console (the appearance and layout will vary based on the browser used)*

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.

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

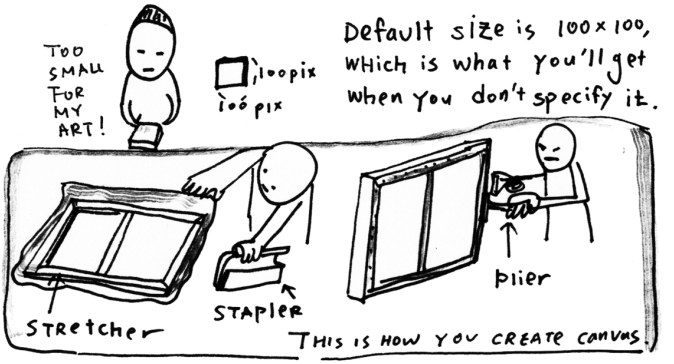
Learning how to program with p5.js involves exploring lots of code: running, altering, breaking, and enhancing it until you have reshaped it into something new. With this in mind, the p5.js website has dozens of examples that demonstrate different features of the library. Visit the Examples page to see 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.

createCanvas() is a function that creates a drawing surface and attach it to the html page.



WRITE it like this.

```
function setup(){
  createCanvas(1920, 1080);
}
```

AND YOU CAN KICK BACK AND CODE WHILE LOOKING at HD TV.

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.

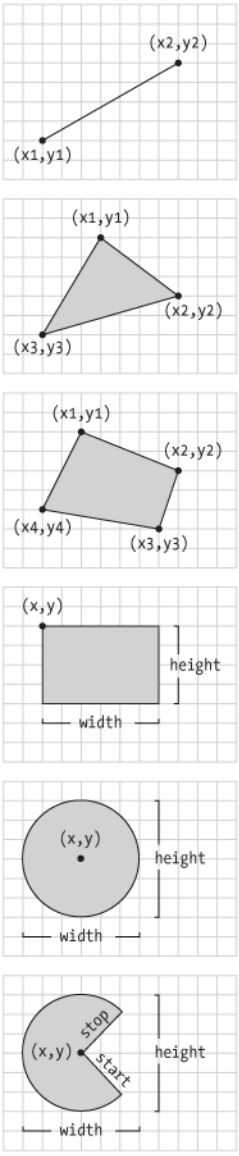


Figure 3-1. *Shapes and their coordinates*

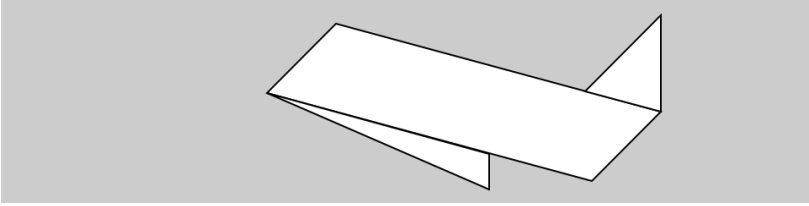
Example 3-3: Draw a Line

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

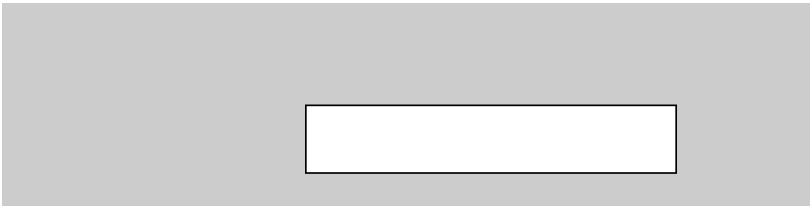
Following this pattern, a triangle needs six parameters and a quadrilateral needs eight (one pair for each point):



```
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

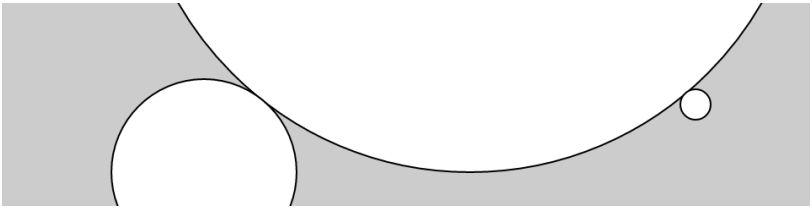
Rectangles and *ellipses* are both defined with four parameters: the first and second are the *x* and *y* coordinates of the anchor point, the third for the width, and the fourth for the height. To make a rectangle at coordinate (180, 60) with a width of 220 pixels and height of 40, use the `rect()`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



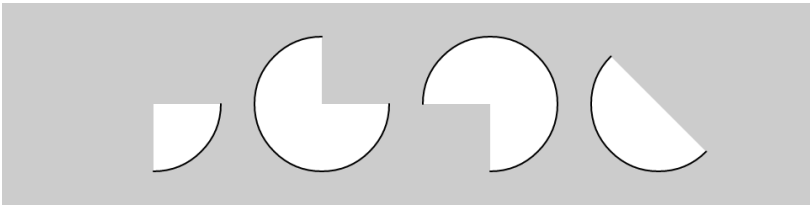
```
function setup() {
  createCanvas(480, 120);
}
```

```
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°.

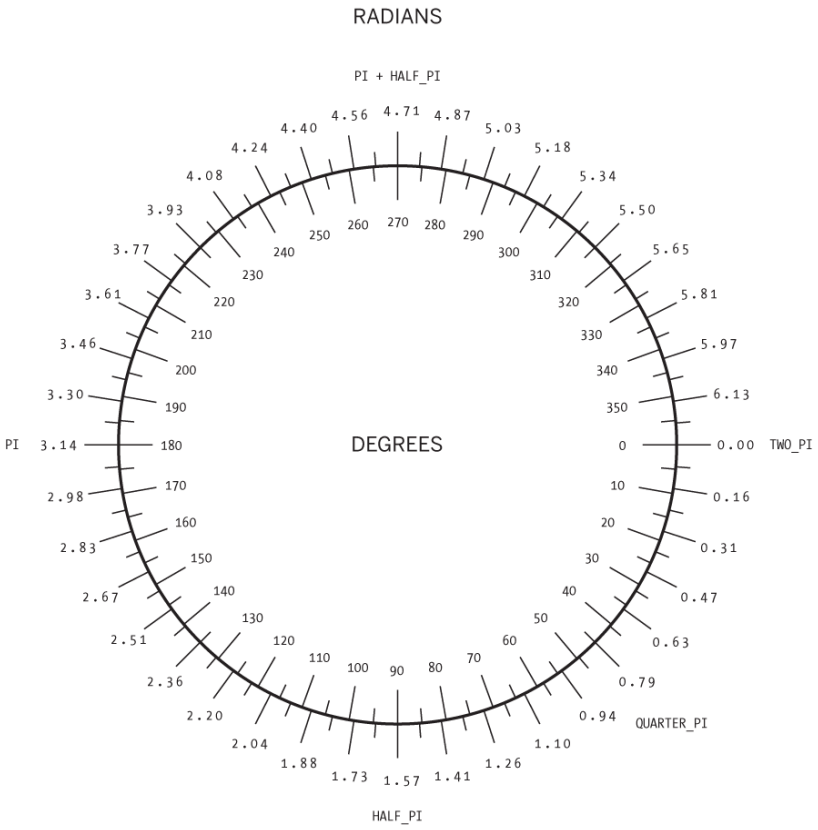


Figure 3-2. *Radians and degrees are two ways to measure an angle. Degrees move around the circle from 0 to 360, while radians measure the angles in relation to pi, from 0 to approximately 6.28.*

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 on page 23, 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 on page 24, 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);
}
```

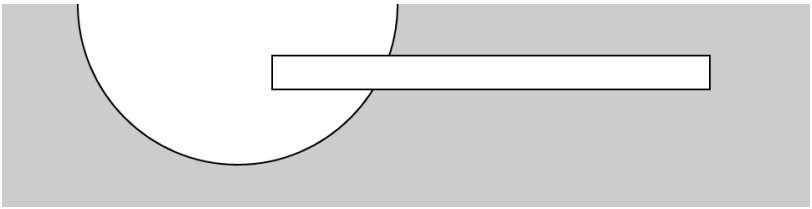
Drawing Order

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.

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.

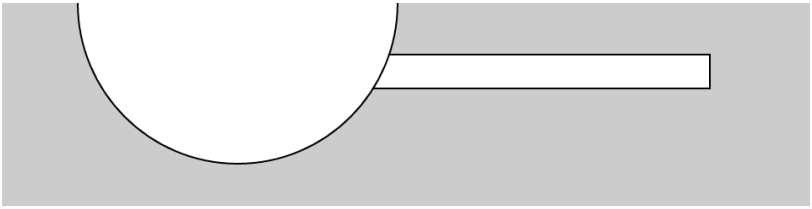
Example 3-10: Control Your Drawing Order



```
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);
}
```

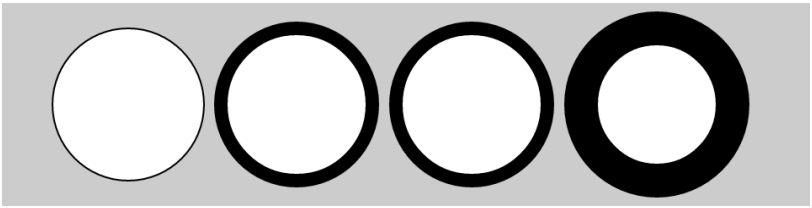
You can think of it like painting with a brush or making a collage. The last element that you add is what's visible on top.

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

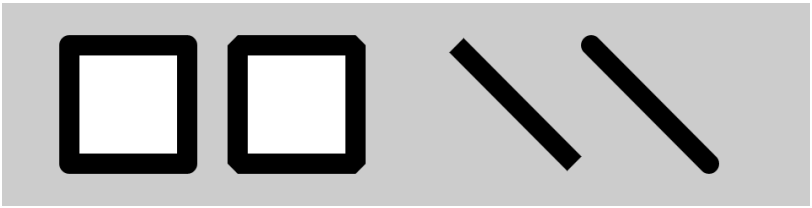
The default stroke weight is a single pixel, but this can be changed with the `strokeWeight()` function. The single parameter to `strokeWeight()` sets the width of drawn lines:



```
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 on page 28, 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.

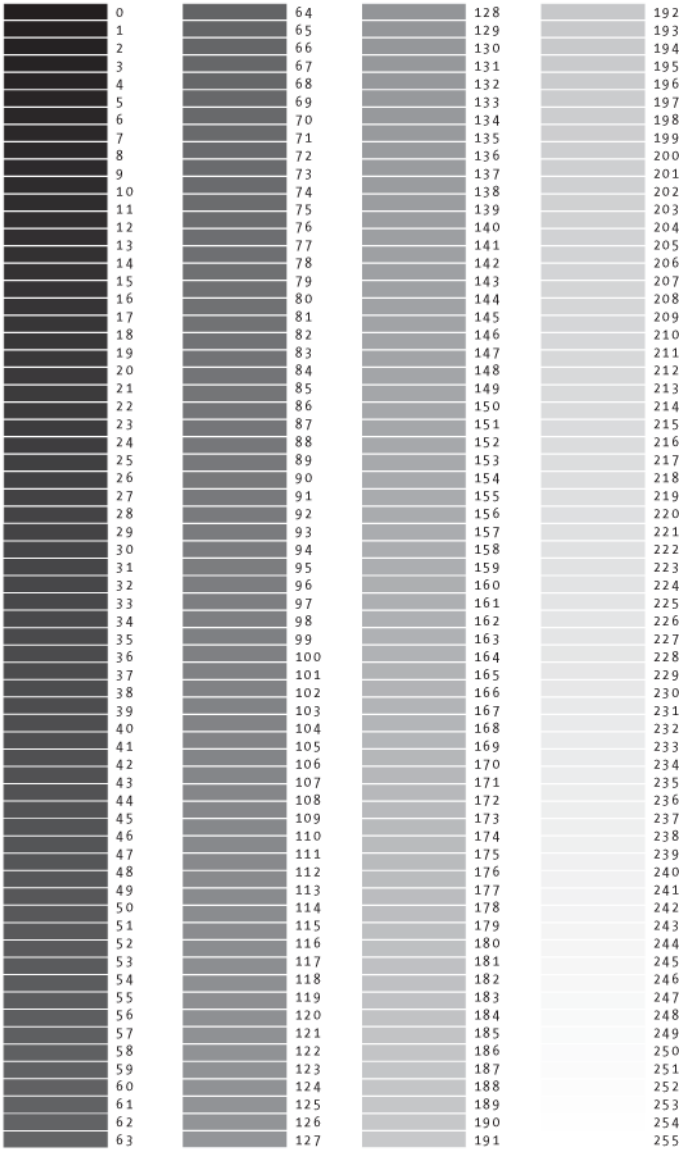
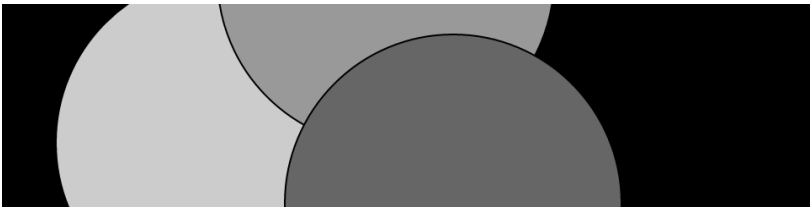


Figure 3-3. *Gray values from 0 to 255*

Example 3-14: Paint with Grays

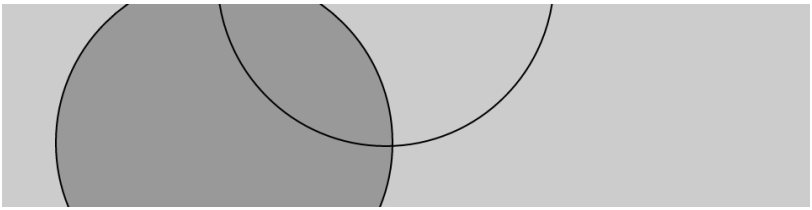
This example shows three different gray values on a black background:



```
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()`:



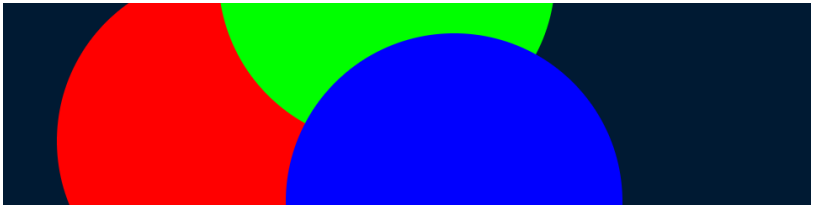
```
function setup() {
  createCanvas(480, 120);
}

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:



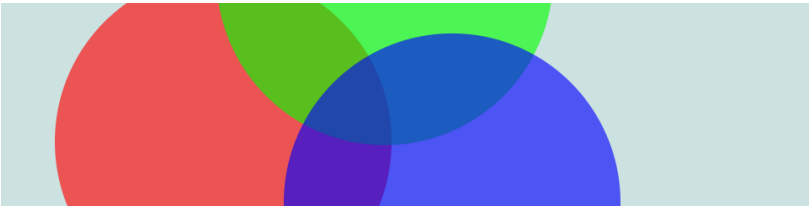
```
function setup() {
  createCanvas(480, 120);
  noStroke();
}

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:



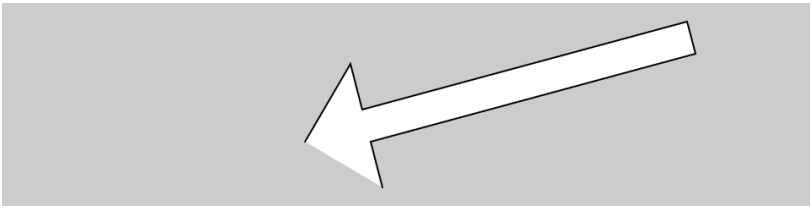
```
function setup() {
  createCanvas(480, 120);
  noStroke();
}

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

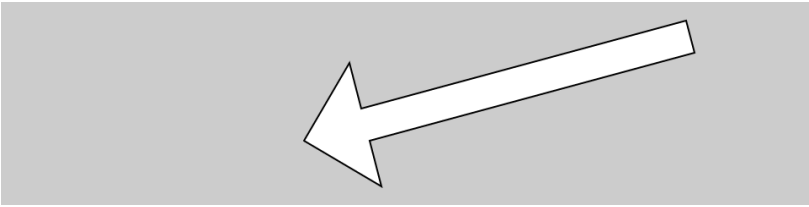
The `beginShape()` function signals the start of a new shape. The `vertex()` function is used to define each pair of *x* and *y* coordinates for the shape. Finally, `endShape()` is called to signal that the shape is finished:



```
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

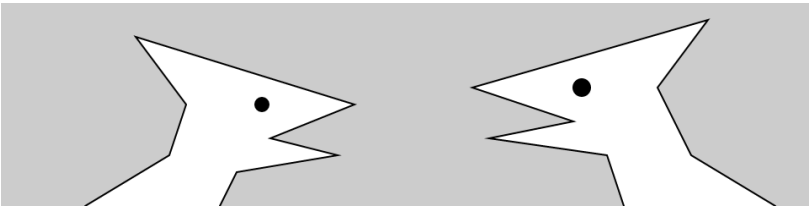
When you run Example 3-18 on page 34, you'll see the first and last point are not connected. To do this, add the word `CLOSE` as a parameter to `endShape()`, like this:



```
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:

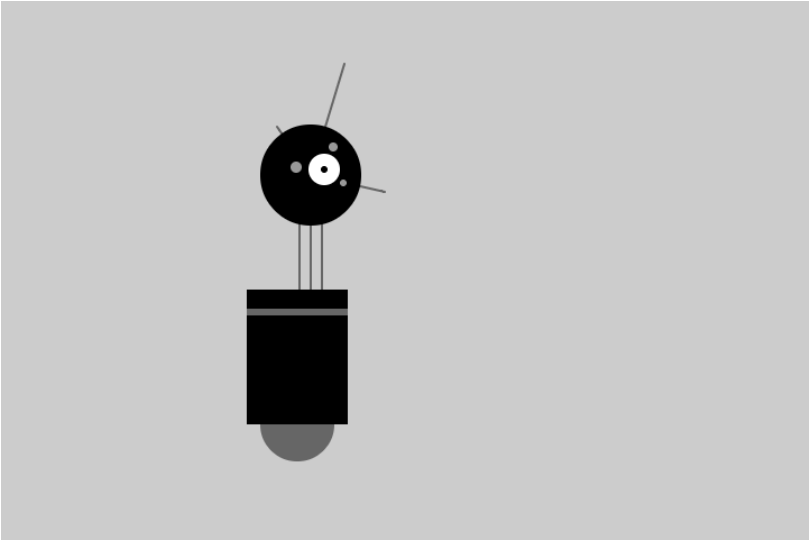
```
function setup() {
  createCanvas(200, 200);
}

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

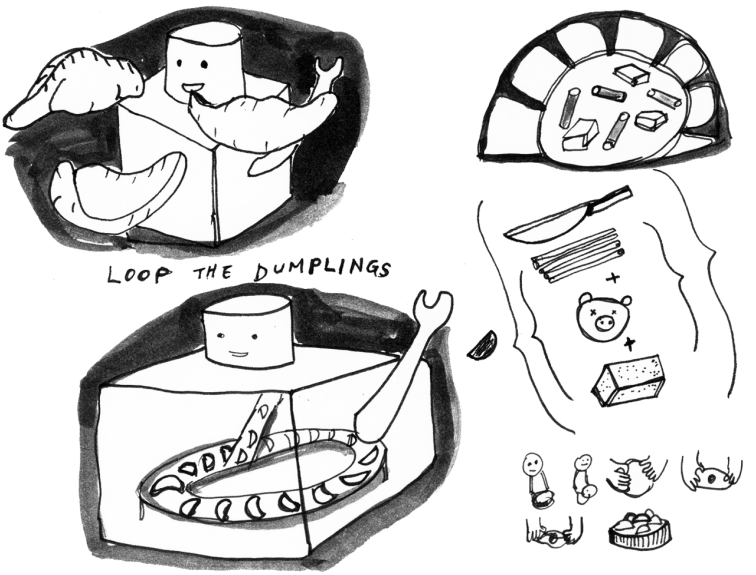


This is P5, the p5.js Robot. There are 10 different programs to draw and animate her in this book—each one explores a different programming idea. P5's design was inspired by Sputnik I (1957), Shakey from the Stanford Research Institute (1966–1972), the fighter drone in David Lynch's *Dune* (1984), and 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
}
```


LOOP THE DUMPLINGS