Chapter 4. Variables

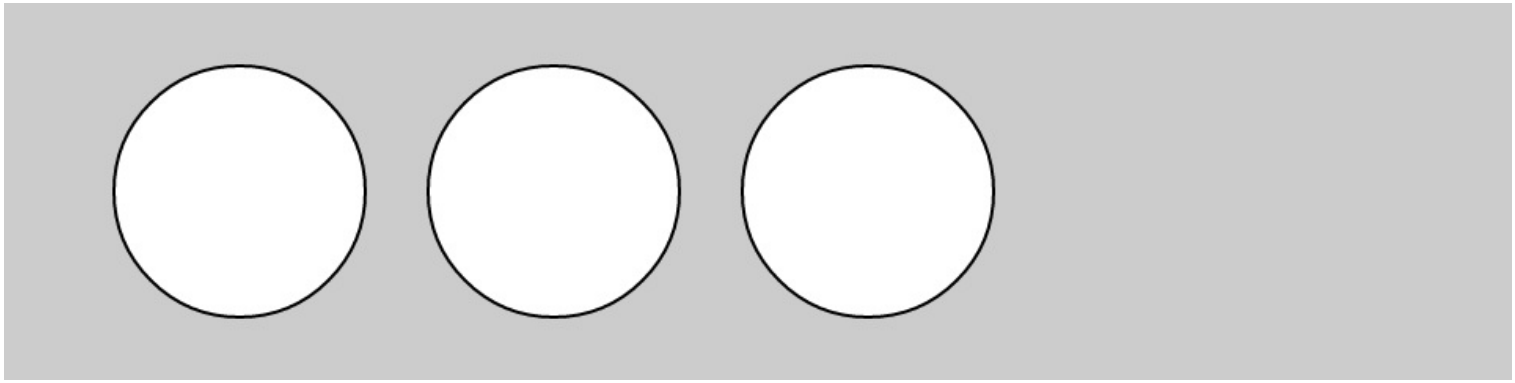
A *variable* stores a value in memory so that it can be used later in a program. A variable can be used many times within a single program, and the value is easily changed while the program is running.

First Variables

The primary reason we use variables is to avoid repeating ourselves in the code. If you are typing the same number more than once, consider using a variable instead so that your code is more general and easier to update.

Example 4-1: Reuse the Same Values

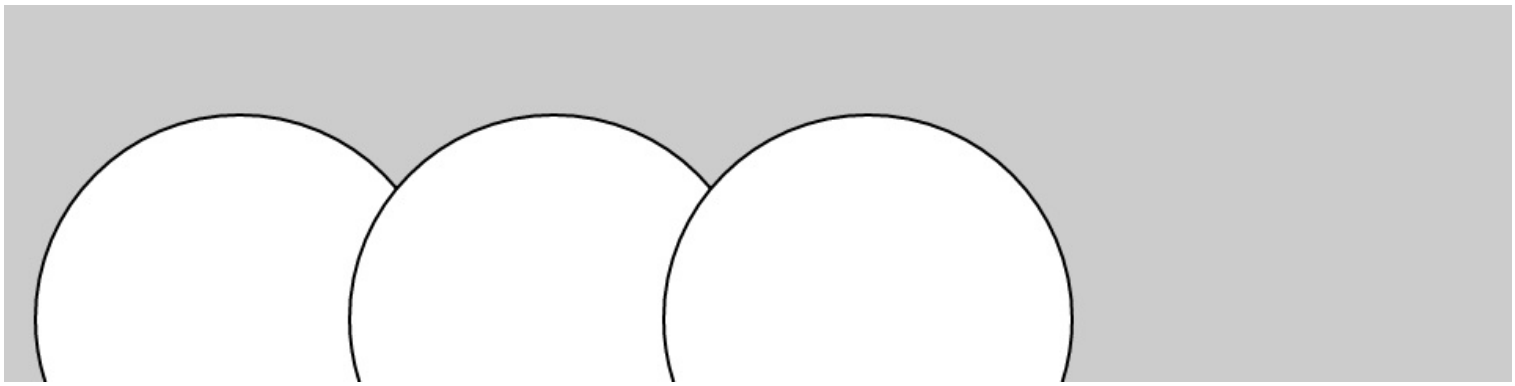
For instance, when you make the *y* coordinate and diameter for the three circles in this example into variables, the same values are used for each ellipse:



```
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
}
```

Example 4-2: Change Values

Simply changing the *y* and *d* variables therefore alters all three ellipses:



```
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
}
```

Without the variables, you'd need to change the *y* coordinate used in the code three times and the diameter six times. When comparing Examples 4-1 and 4-2, notice how all the lines are the same, except the first two lines with the variables are different. Variables allow you to separate the lines of the code that change from the lines that don't, which makes programs easier to modify. For instance, if you place variables that control colors and sizes of shapes in one place, then you can quickly explore different visual options by focusing on only a few lines of code.

Making Variables

When you make your own variables, you determine the *name* and the *value*. The name is what you decide to call the variable. Choose a name that is informative about what the variable stores, but be consistent and not too verbose. For instance, the variable name "radius" will be clearer than "r" when you look at the code later.

Variables must first be *declared*, which sets aside space in the computer's memory to store the information. When declaring a variable, you use var, to indicate you are creating a new variable, followed by the name. After the name is set, a value can be assigned to the variable:

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

This code does the same thing, but is shorter:

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

The characters var are included on the line of code that declares a variable, but they're not written again. Each time var is written in front of the variable name, the computer thinks you're trying to declare a new variable. You can't have two variables with the same name in the same part of the program (Appendix C), or the program could behave strangely:

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

You can place your variables outside of setup() and draw(). If you create a variable inside of setup(), you can't use it inside of

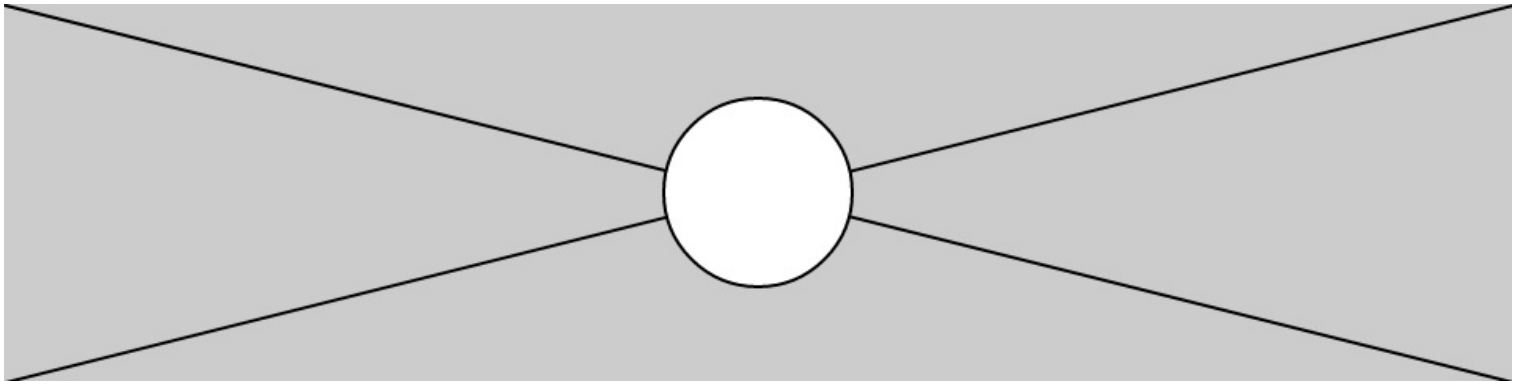
draw(), so you need to place those variables somewhere else. Such variables are called *global variables*, because they can be used anywhere ("globally") in the program.

p5.js Variables

p5.js has a series of special variables to store information about the program while it runs. For instance, the width and height of the canvas are stored in variables called width and height. These values are set by the createCanvas() function. They can be used to draw elements relative to the size of the canvas, even if the createCanvas() line changes.

Example 4-3: Adjust the Canvas, See What Follows

In this example, change the parameters to createCanvas() to see how it works:



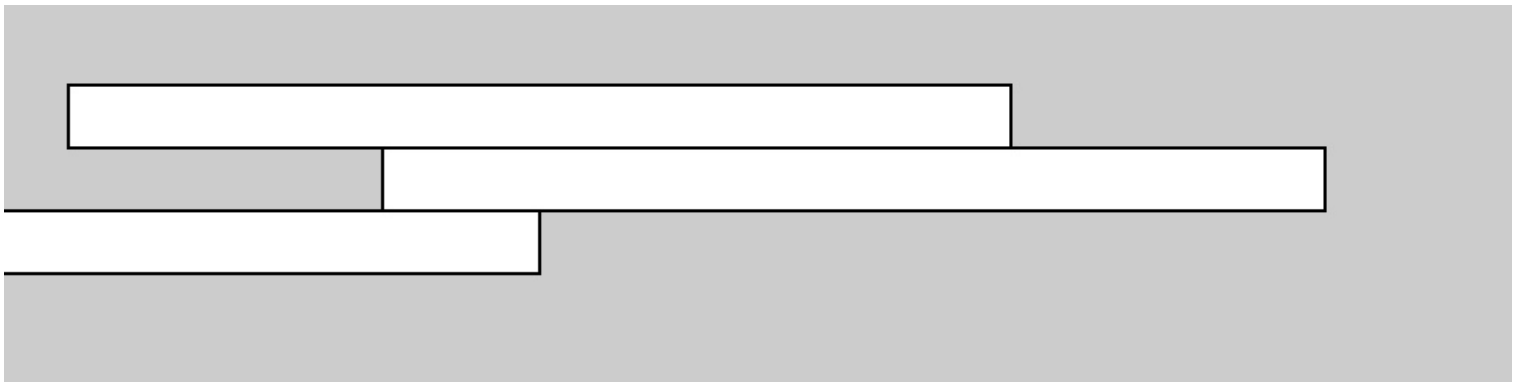
```
function setup() {
  createCanvas(480, 120);
}

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);
}
```

Other special variables keep track of the status of the mouse and keyboard values and much more. These are discussed in Chapter 5.

A Little Math

People often assume that math and programming are the same thing. Although knowledge of math can be useful for certain types of coding, basic arithmetic covers the most important parts.

Example 4-4: Basic Arithmetic



```
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
}
```

In code, symbols like +, −, and * are called *operators*. When placed between two values, they create an *expression*. For instance, 5 + 9 and 1024 − 512 are both expressions. The operators for the basic math operations are:

+	Addition
−	Subtraction
*	Multiplication
/	Division
=	Assignment

JavaScript has a set of rules to define which operators take precedence over others, meaning which calculations are made first, second, third, and so on. These rules define the order in which the code is run. A little knowledge about this goes a long way toward understanding how a short line of code like this works:

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

The expression 4 * 5 is evaluated first because multiplication has the highest priority. Second, 4 is added to the product of 4 * 5 to yield 24. Last, because the *assignment operator* (the *equals* sign) has the lowest precedence, the value 24 is assigned to the variable *x*. This is clarified with parentheses, but the result is the same:

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

If you want to force the addition to happen first, just move the parentheses. Because parentheses have a higher precedence than multiplication, the order is changed and the calculation is affected:

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

An acronym for this order is often taught in math class: PEMDAS, which stands for Parentheses, Exponents, Multiplication, Division, Addition, Subtraction, where parentheses have the highest priority and subtraction the lowest. The complete order of operations is found in Appendix B.

Some calculations are used so frequently in programming that shortcuts have been developed; it's always nice to save a few keystrokes. For instance, you can add to a variable, or subtract from it, with a single operator:

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

It's also common to add or subtract 1 from a variable, so shortcuts exist for this as well. The ++ and -- operators do this:

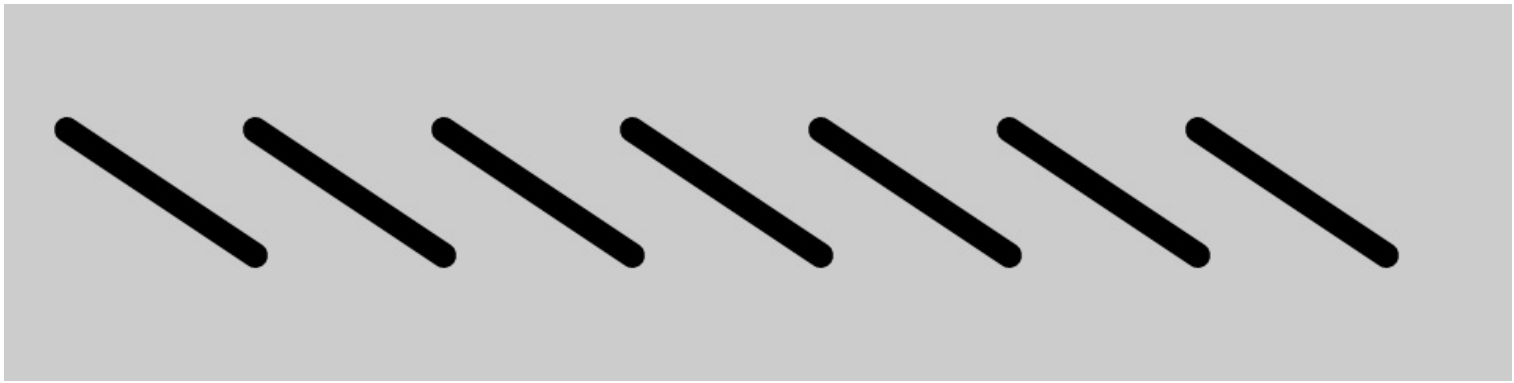
```
x++; // This is the same as x = x + 1
y--; // This is the same as y = y - 1
```

Repetition

As you write more programs, you'll notice that patterns occur when lines of code are repeated, but with slight variations. A code structure called a for loop makes it possible to run a line of code more than once to condense this type of repetition into fewer lines. This makes your programs more modular and easier to change.

Example 4-5: Do the Same Thing Over and Over

This example has the type of pattern that can be simplified with a for loop:



```
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);
}
```

Example 4-6: Use a for Loop

The same thing can be done with a for loop, and with less code:

```
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);
  }
}
```

The for loop is different in many ways from the code we've written so far. Notice the braces, the { and } characters. The code between the braces is called a *block*. This is the code that will be repeated on each iteration of the for loop.

Inside the parentheses are three statements, separated by semicolons, that work together to control how many times the code inside the block is run. From left to right, these statements are referred to as the *initialization* (init), the *test*, and the *update*:

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

The init typically declares a new variable to use within the for loop and assigns a value. The variable name i is frequently used, but there's really nothing special about it. The *test* evaluates the value of this variable, and the *update* changes the variable's value. Figure 4-1 shows the order in which they run and how they control the code statements inside the block.

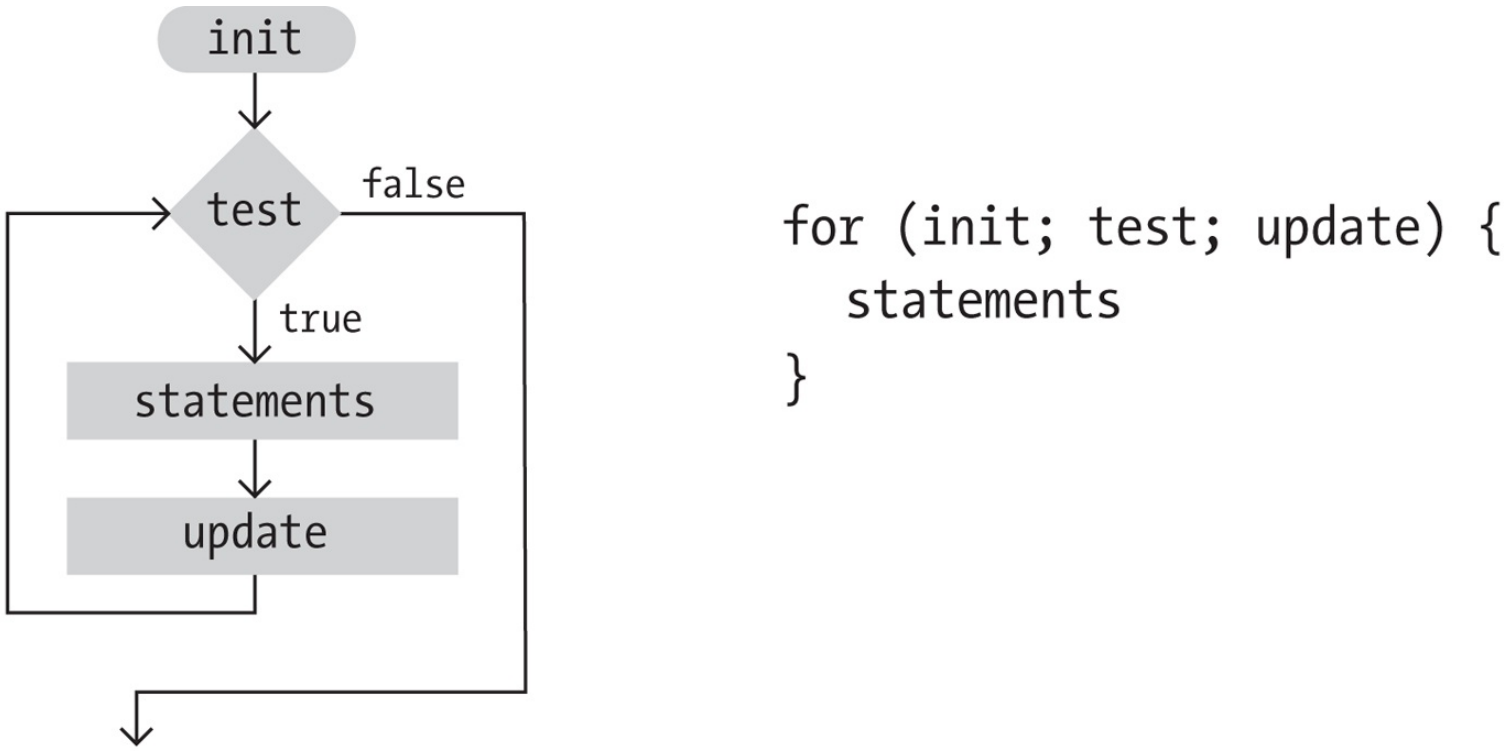


Figure 4-1. Flow diagram of a for loop

The *test* statement requires more explanation. It's always a *relational expression* that compares two values with a *relational operator*. In this example, the expression is "i < 400" and the operator is the < (less than) symbol. The most common relational operators are:

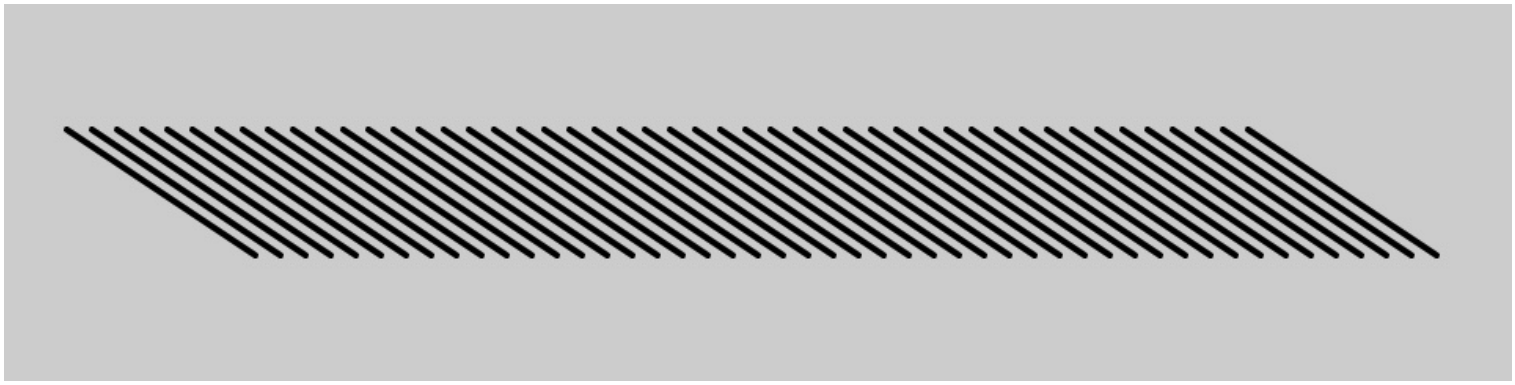
>	Greater than
<	Less than
>=	Greater than or equal to
<=	Less than or equal to
==	Equal to

!=	Not equal to

The relational expression always evaluates to true or false. For instance, the expression 5 > 3 is true. We can ask the question, "Is five greater than three?" Because the answer is "yes," we say the expression is true. For the expression 5 < 3, we ask, "Is five less than three?" Because the answer is "no," we say the expression is false. When the evaluation is true, the code inside the block is run, and when it's false, the code inside the block is not run and the for loop ends.

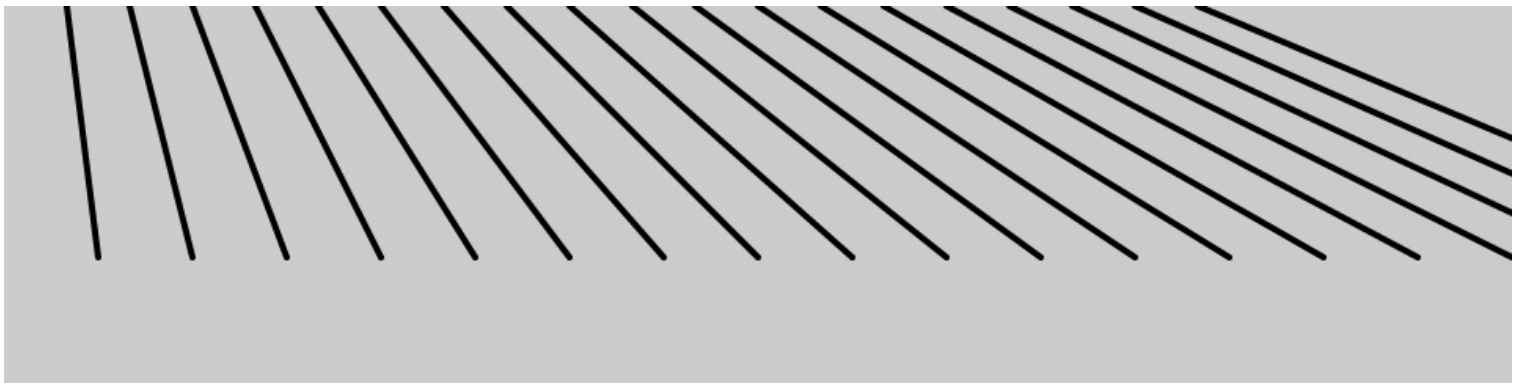
Example 4-7: Flex Your for Loop's Muscles

The ultimate power of working with a for loop is the ability to make quick changes to the code. Because the code inside the block is typically run multiple times, a change to the block is magnified when the code is run. By modifying Example 4-6 only slightly, we can create a range of different patterns:



```
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);
  }
}
```

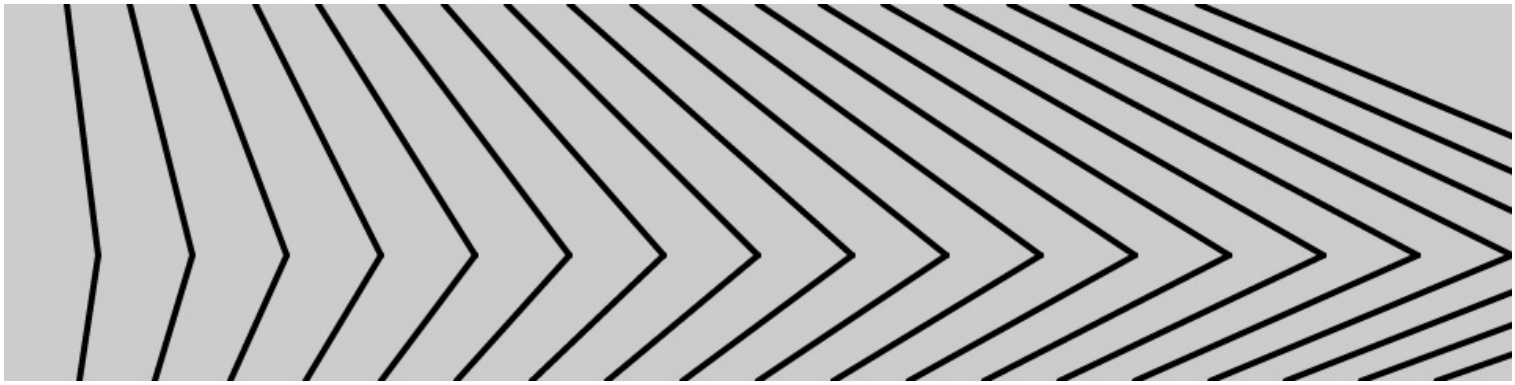
Example 4-8: Fanning Out the Lines



```
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);
  }
}
```

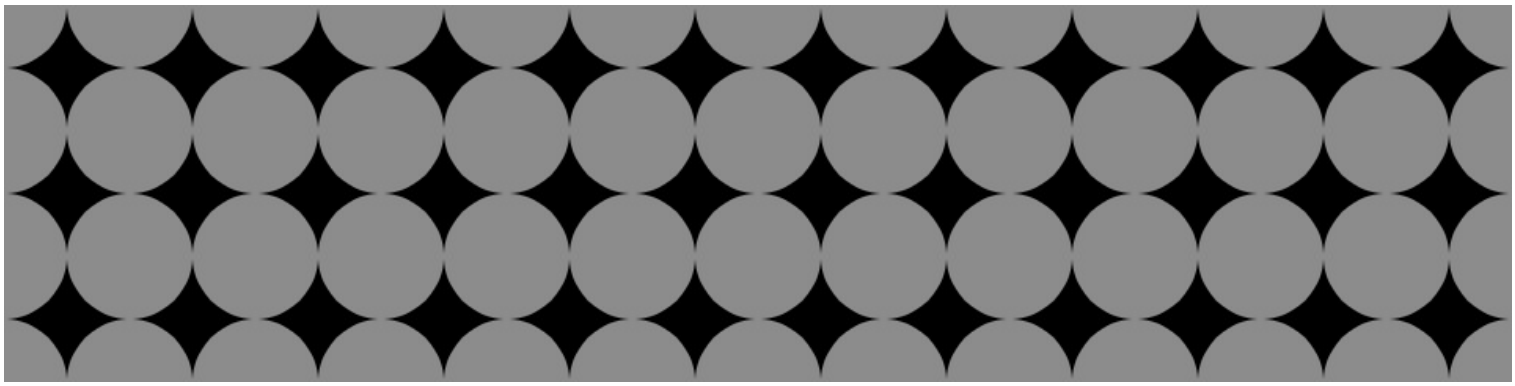
Example 4-9: Kinking the Lines



```
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);
  }
}
```

Example 4-10: Embed One for Loop in Another

When one for loop is embedded inside another, the number of repetitions is multiplied. First, let's look at a short example, and then we'll break it down in Example 4-11:

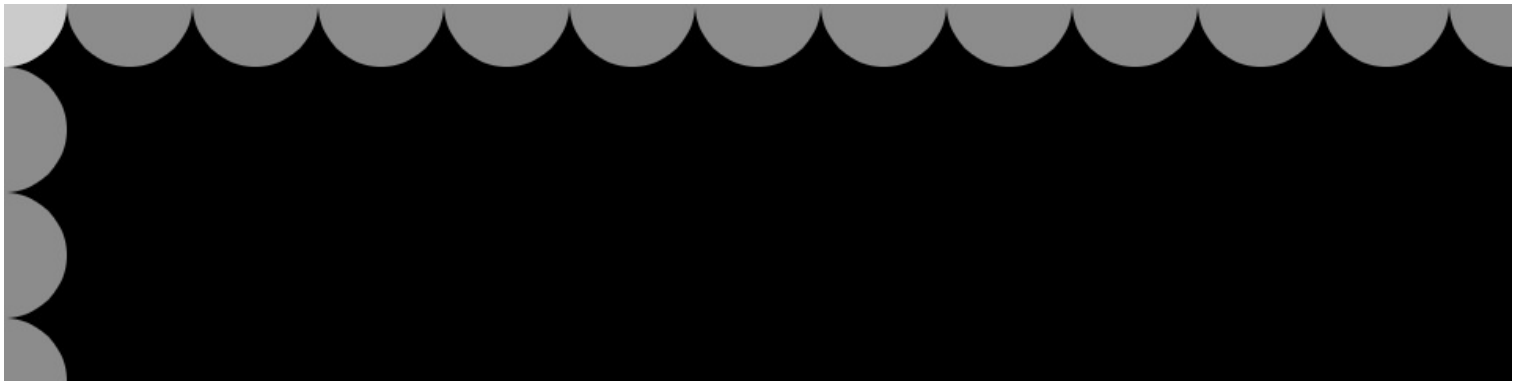


```
function setup() {
  createCanvas(480, 120);
  noStroke();
}

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);
    }
  }
}
```

Example 4-11: Rows and Columns

In this example, the for loops are adjacent, rather than one embedded inside the other. The result shows that one for loop is drawing a column of 4 circles and the other is drawing a row of 13 circles:



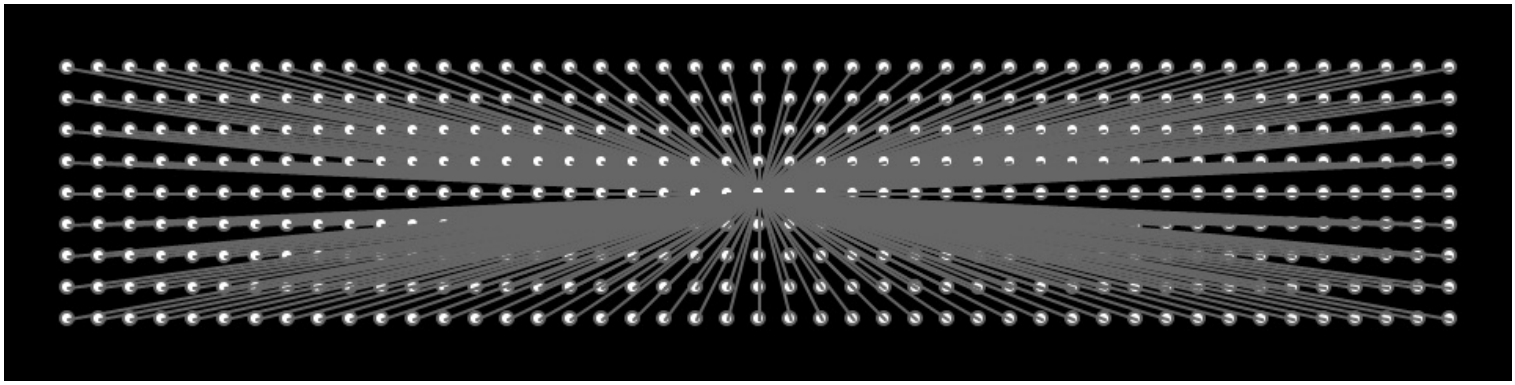
```
function setup() {
  createCanvas(480, 120);
  noStroke();
}

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);
  }
}
```

When one of these for loops is placed inside the other, as in Example 4-10, the 4 repetitions of the first loop are compounded with the 13 of the second in order to run the code inside the embedded block 52 times (4×13 = 52).

Example 4-10 is a good base for exploring many types of repeating visual patterns. The following examples show a couple of ways that it can be extended, but this is only a tiny sample of what's possible.

Example 4-12: Pins and Lines

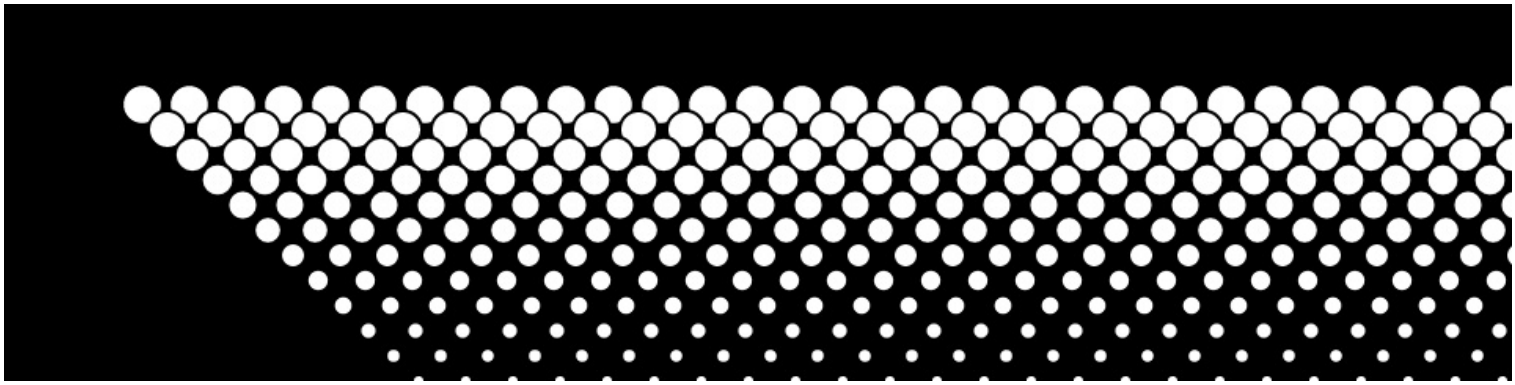
In this example, the code draws a line from each point in the grid to the center of the screen:



```
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);
    }
  }
}
```

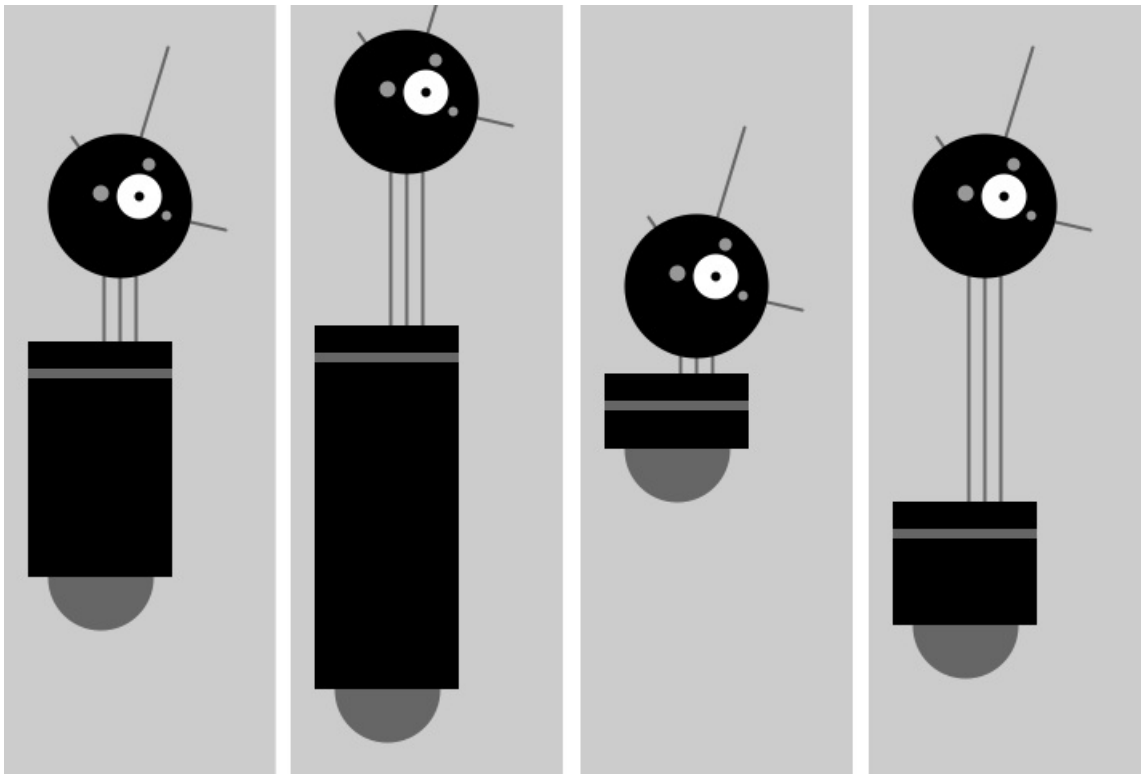
Example 4-13: Halftone Dots

In this example, the ellipses shrink with each new row and are moved to the right by adding the *y* coordinate to the *x* coordinate:



```
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);
    }
  }
}
```

Robot 2: Variables



The variables introduced in this program make the code look more difficult than Robot 1 (see "Robot 1: Draw"), but now it's much easier to modify, because numbers that depend on one another are in a single location. For instance, the neck is drawn based on the neckHeight variable. The group of variables at the top of the code control the aspects of the robot that we want to change: location, body height, and neck height. You can see some of the range of possible variations in the figure; from left to right, here are the values that correspond to them:

y = 390 bodyHeight = 180 neckHeight = 40	y = 460 bodyHeight = 260 neckHeight = 95	y = 310 bodyHeight = 80 neckHeight = 10	y = 420 bodyHeight = 110 neckHeight = 140

When altering your own code to use variables instead of numbers, plan the changes carefully, then make the modifications in short steps. For instance, when this program was written, each variable was created one at a time to minimize the complexity of the transition. After a variable was added and the code was run to ensure it was working, the next variable was added:

```
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:

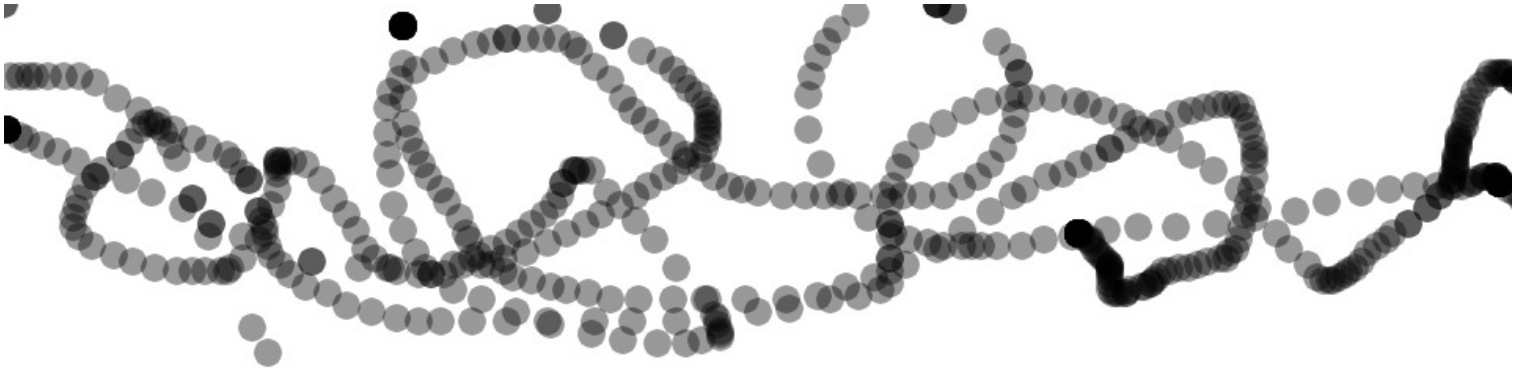
```
var x = 280;
var y = -100;
var diameter = 380;

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

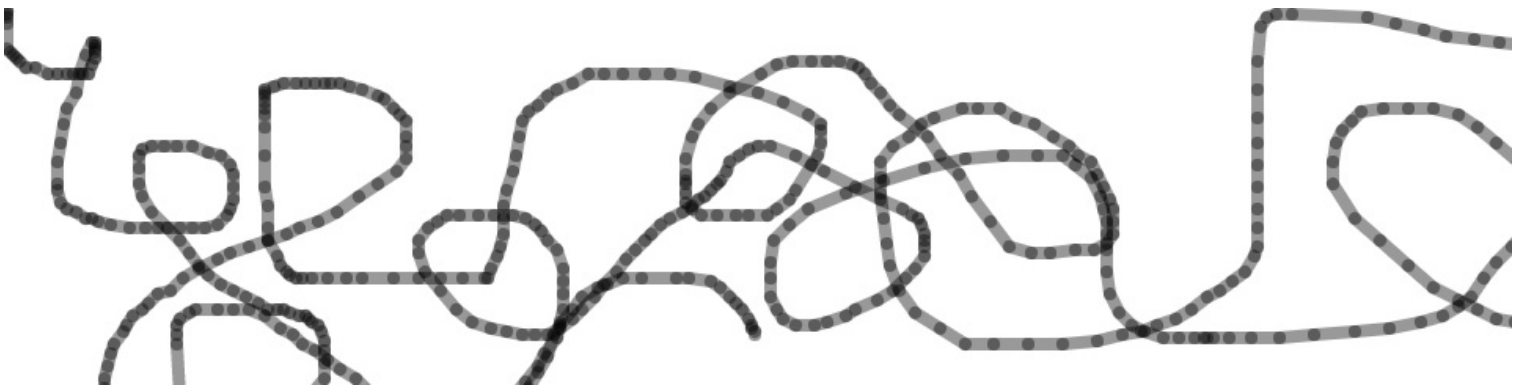
In this example, a new circle is added to the canvas each time the code in draw() is run. To refresh the screen and only display the newest circle, place a background() function at the beginning of draw() before the shape is drawn:

```
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

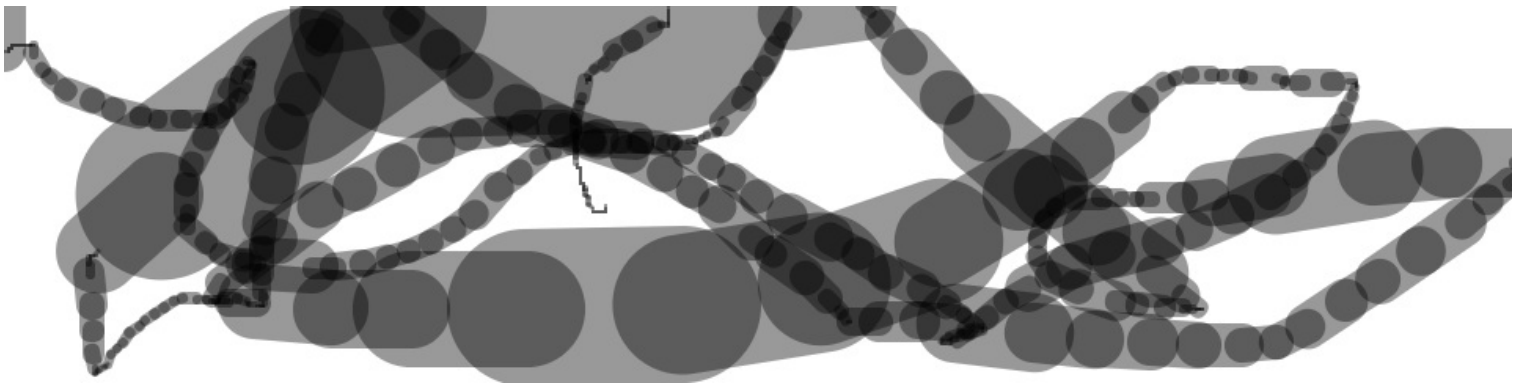
The pmouseX and pmouseY variables store the position of the mouse at the previous frame. Like mouseX and mouseY, these special variables are updated each time draw() runs. When combined, they can be used to draw continuous lines by connecting the current and most recent location:



```
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.

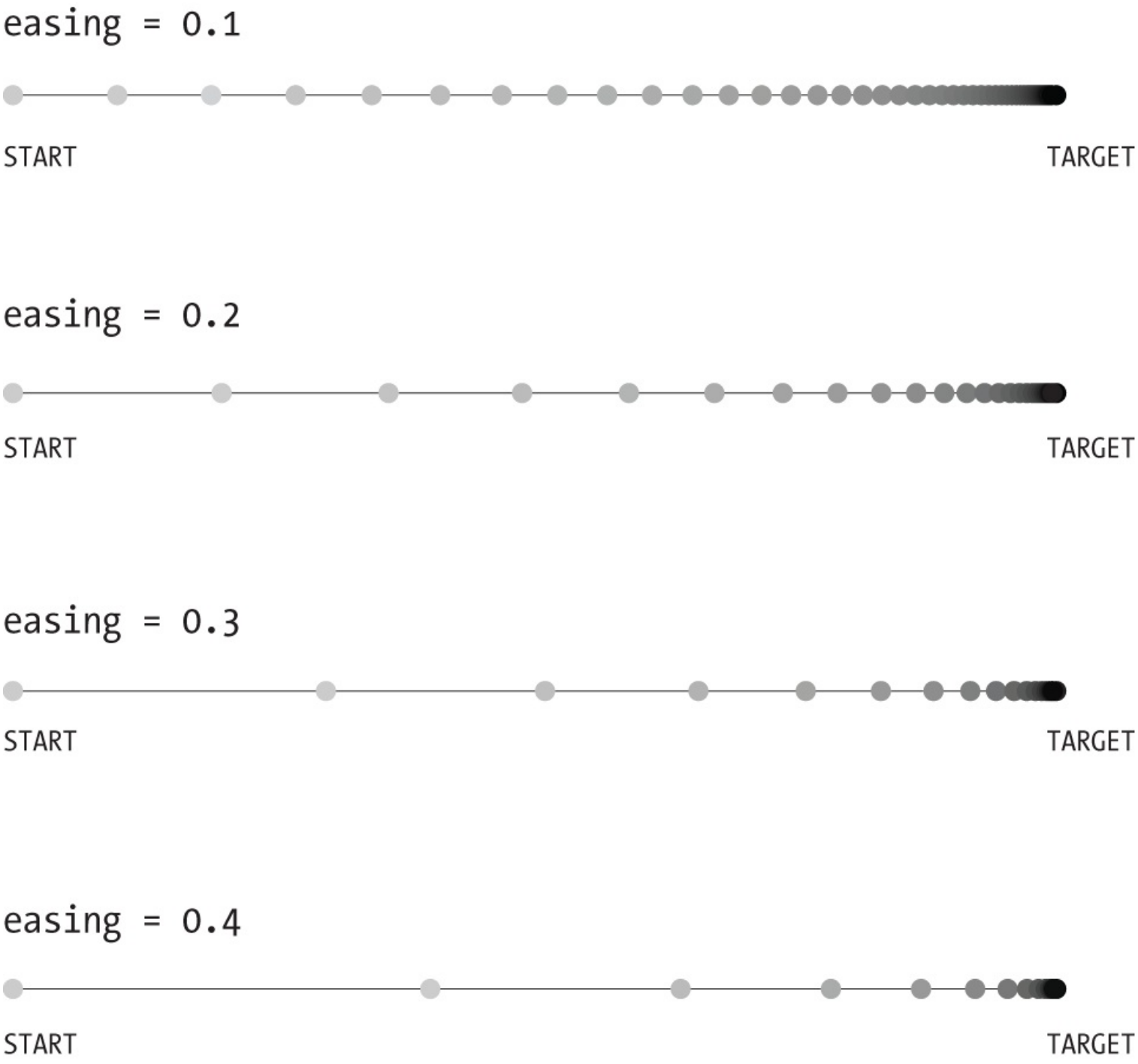
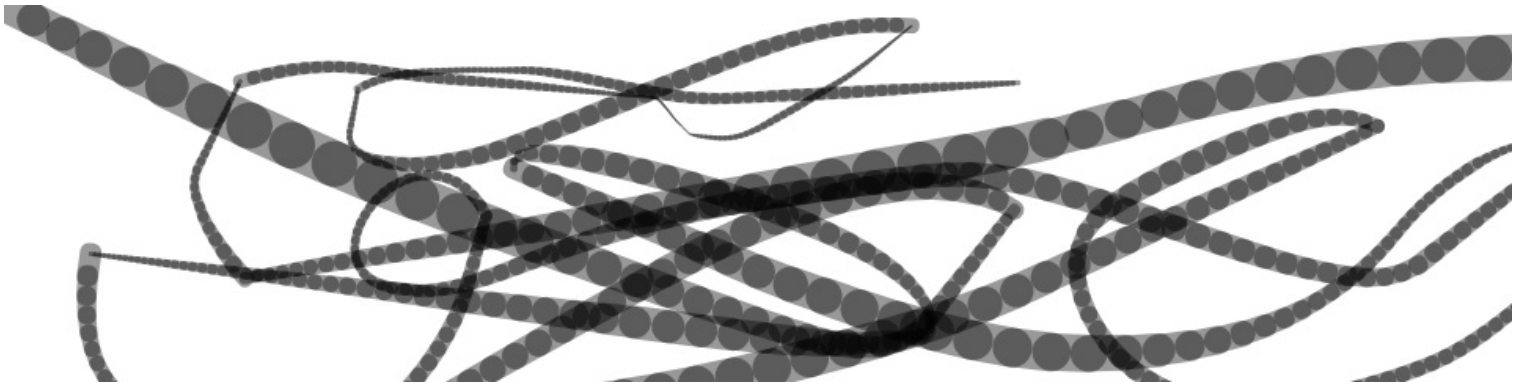


Figure 5-1. Easing changes the number of steps it takes to move from one place to another

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:



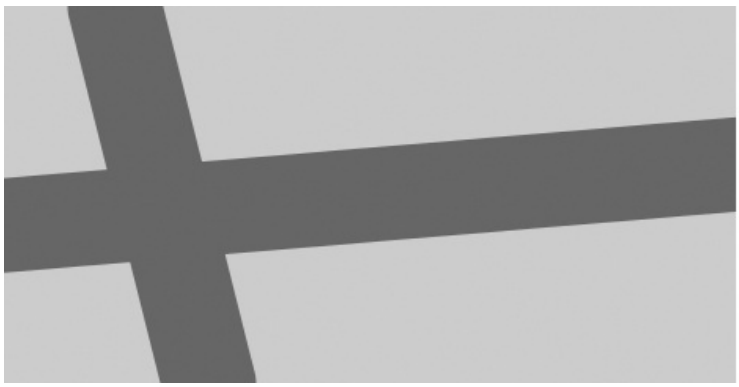
```
var x = 0;
var y = 0;
var px = 0;
var py = 0;
var easing = 0.05;

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;
}
```

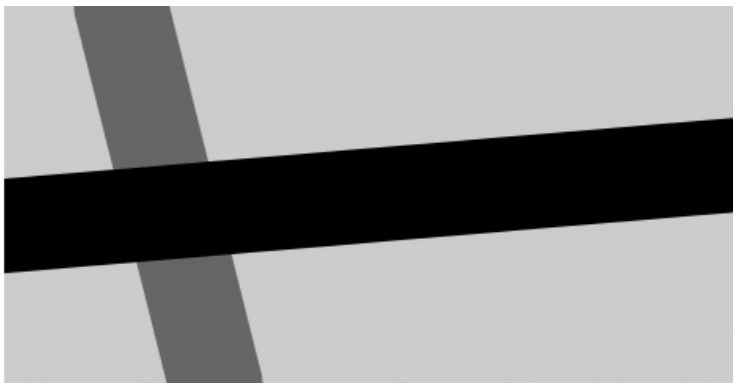
Click

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.

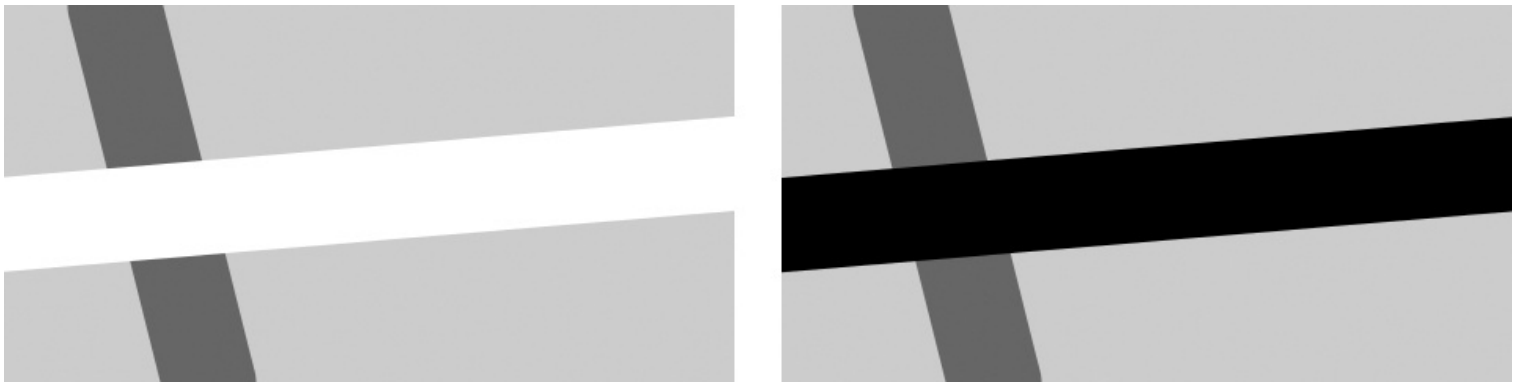
Alternatively, the test in draw() can be written like this:

```
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.

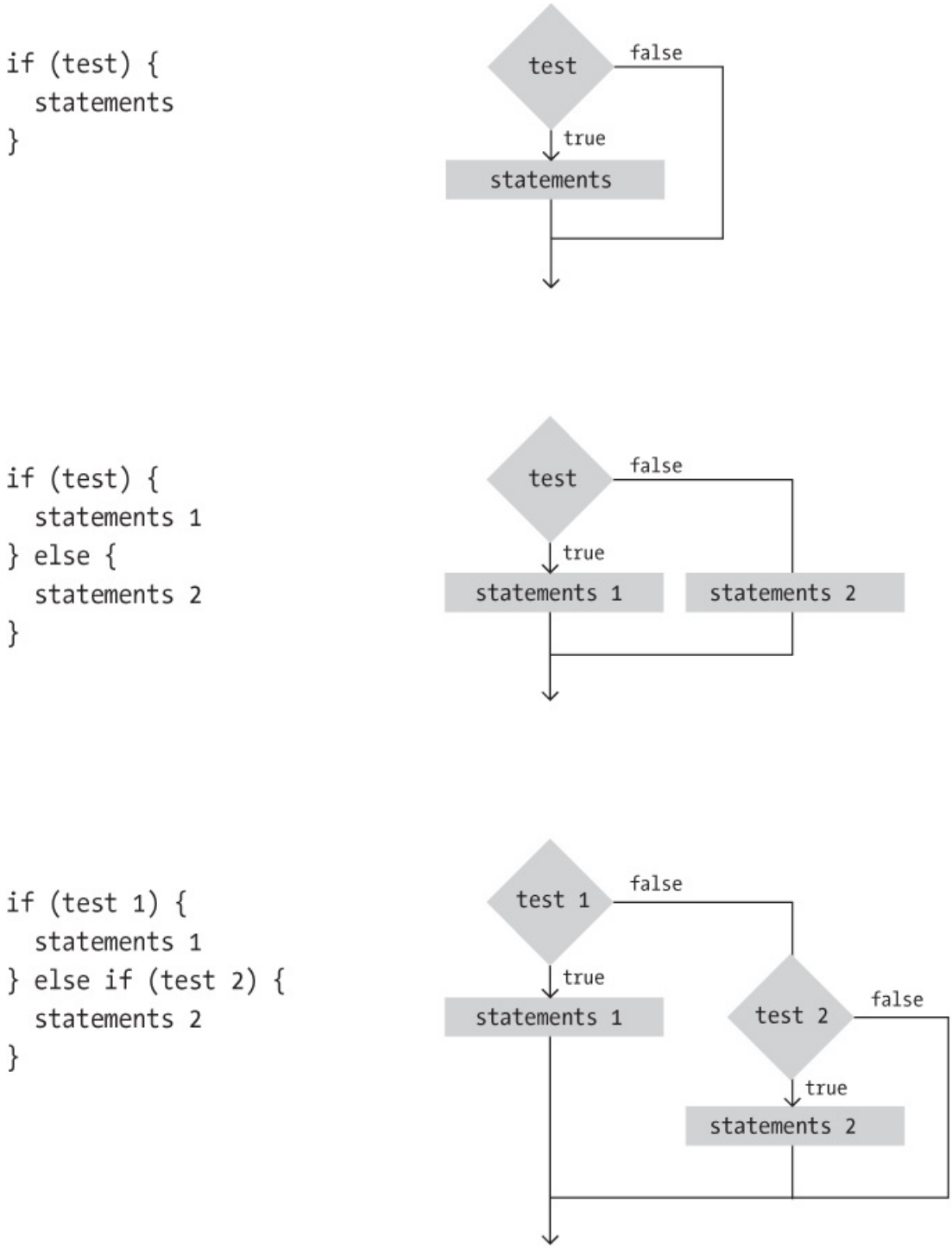


Figure 5-2. The if and else structure makes decisions about which blocks of code to run

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

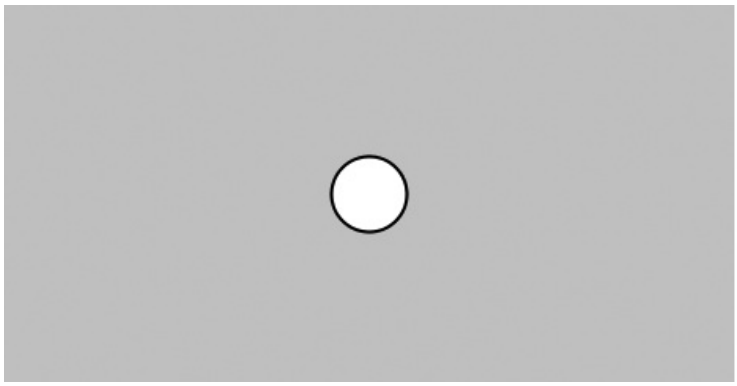
For instance, this example tests to see whether the cursor is on the left or right side of a line and then moves the line toward the cursor:

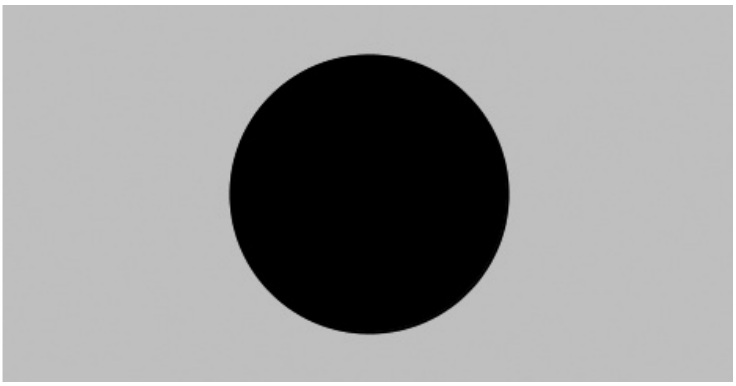
```
var x;
var offset = 10;

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.

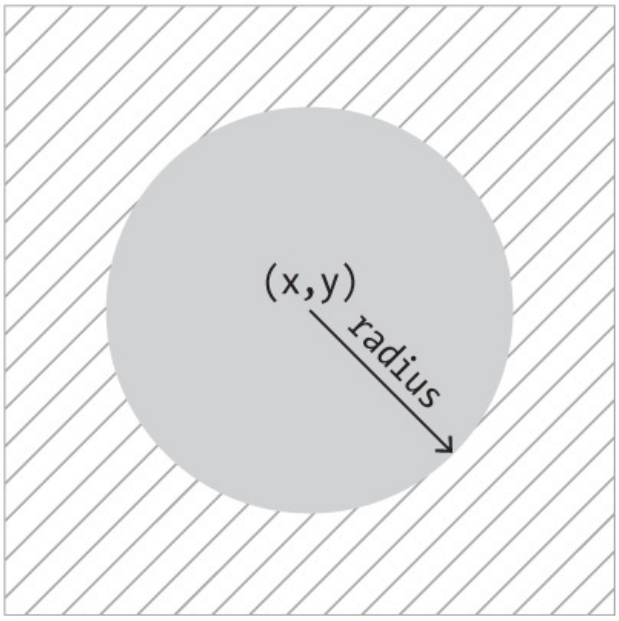
Example 5-14: The Bounds of a Circle

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:

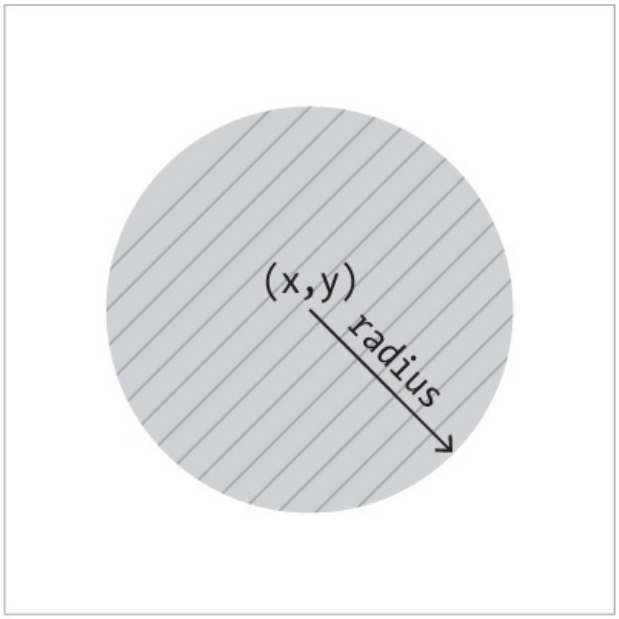




```
var x = 120;
var y = 60;
var radius = 12;

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);
}
```




```
dist(x, y, mouseX, mouseY) > radius
```

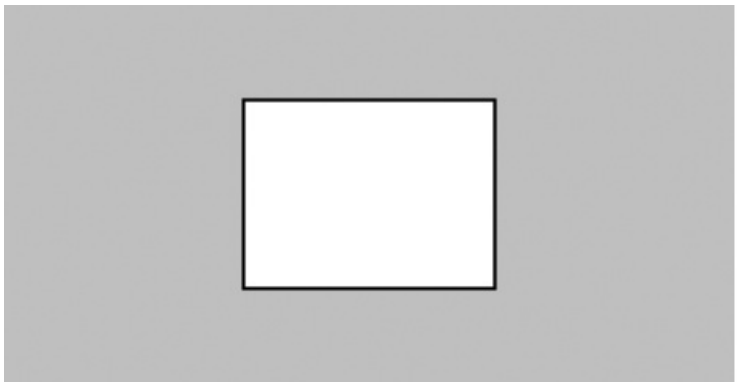



```
dist(x, y, mouseX, mouseY) < radius
```

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.

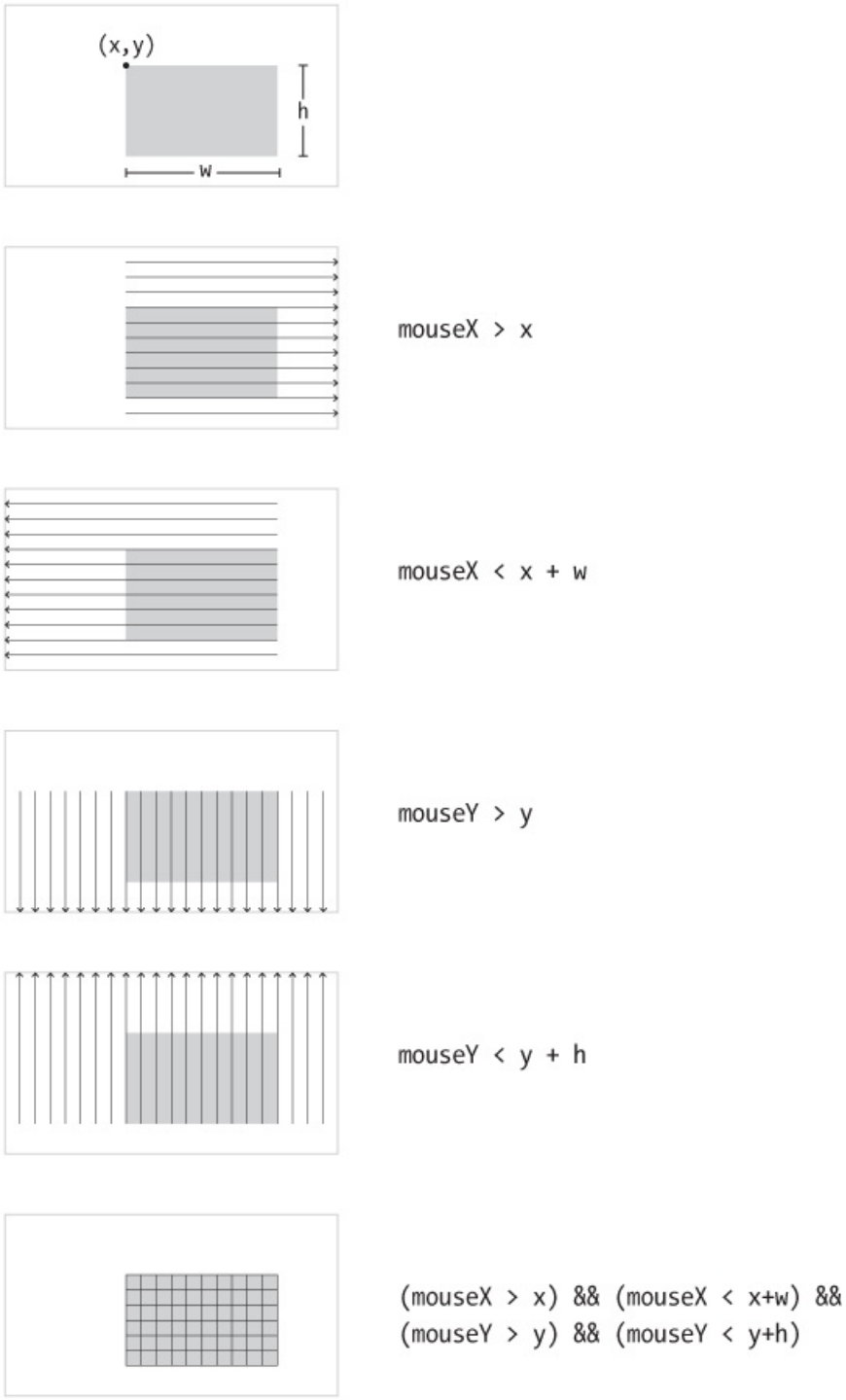


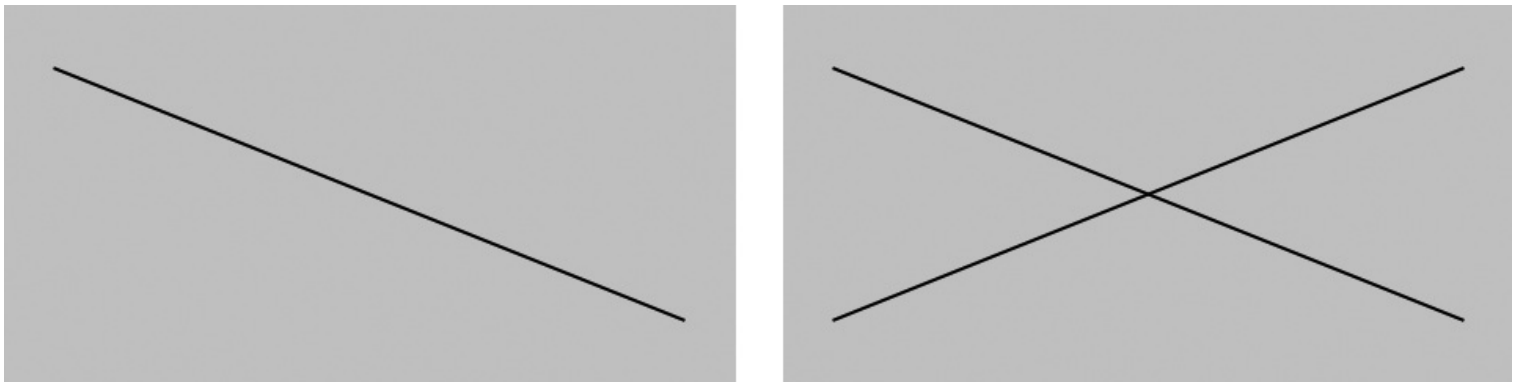
Figure 5-4. Rectangle rollover test. When all four tests are combined and true, the cursor is inside the rectangle.

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

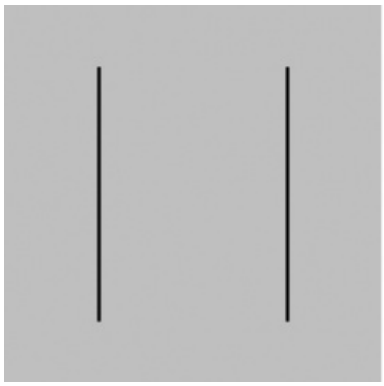
This example introduces the textSize() function to set the size of the letters, the textAlign() function to center the text on its *x* coordinate, and the text() function to draw the letter. These functions are discussed in more detail in "Fonts".

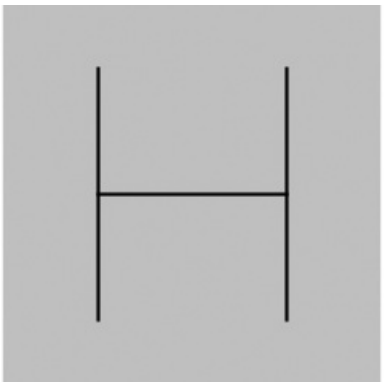

```
function setup() {
  createCanvas(120, 120);
  textSize(64);
  textAlign(CENTER);
  fill(255);
}

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.

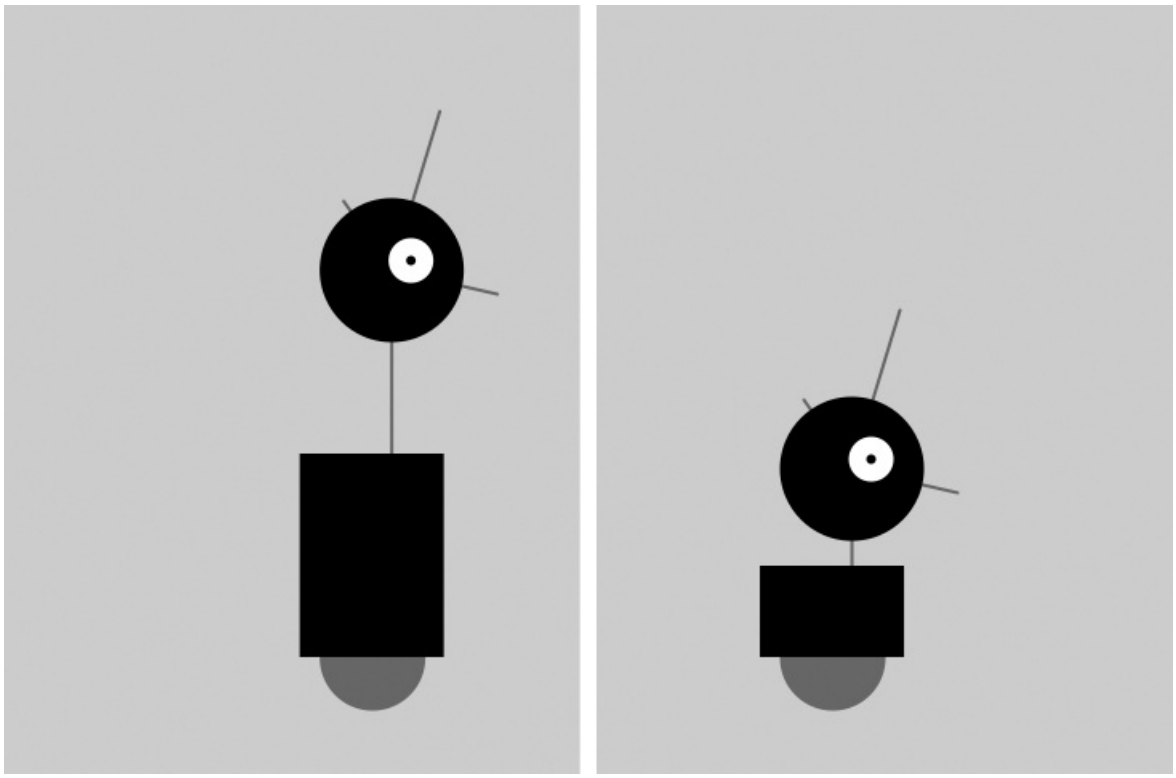
Example 5-23: Map with the map() Function

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);
}
```