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 Processing and make your first sketch.

Start by visiting *http://processing.org/download* and selecting the Mac, Windows, or Linux version, depending on what machine you have. Installation on each machine is straightforward:

- On Windows, you'll have a *.zip* file. Double-click it, and drag the folder inside to a location on your hard disk. It could be *Program Files* or simply the desktop, but the important thing is for the *processing* folder to be pulled out of that *.zip* file. Then double-click *processing.exe* to start.
- The Mac OS X version is a *.zip* file. Double-click it, and drag the Processing icon to the *Applications* folder. If you're using someone else's machine and can't modify the *Applications* folder, just drag the application to the desktop. Then double-click the Processing icon to start.
- The Linux version is a *.tar.gz* file, which should be familiar to most Linux users. Download the file to your home directory, then open a terminal window, and type:

```
tar xvfz processing-xxxx.tgz
```

(Replace *xxxx* with the rest of the file's name, which is the version number.) This will create a folder named *processing-3.0* or something similar. Then change to that directory:

```
    cd processing-xxxx
```

and run it:

```
    ./processing
```

With any luck, the main Processing window will now be visible (Figure 2-1). Everyone's setup is different, so if the program didn't start, or you're otherwise stuck, visit the troubleshooting page for possible solutions.

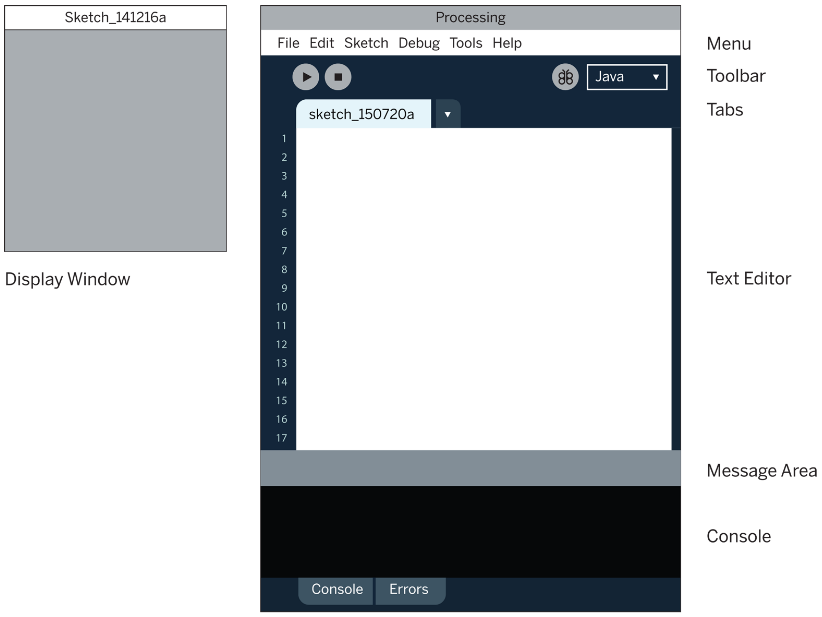


Figure 2-1. *The Processing Development Environment*

Your First Program

You're now running the Processing Development Environment (or PDE). There's not much to it; the large area is the Text Editor, and there's two buttons across the top; this is the Toolbar. Below the editor is the Message Area, and below that is the Console. The Message Area is used for one-line messages, and the Console is used for more technical details.

Example 2-1: Draw an Ellipse

In the editor, type the following:

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

This 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." Click the Run button the (triangle button in the Toolbar).

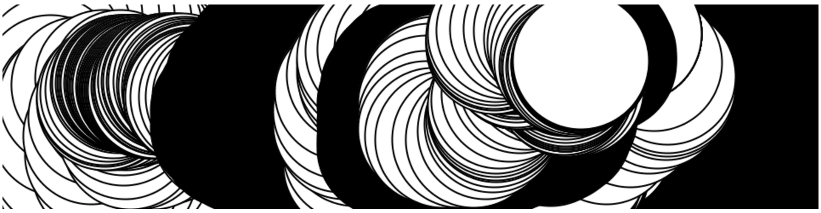
If you've typed everything correctly, you'll see a circle on your screen. If you didn't type it correctly, the Message Area will turn red and complain about an error. If this happens, make sure that you've copied the example code exactly: the numbers should be contained within parentheses and have commas between each of them, and 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 Processing 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:



```
void setup() {
  size(480, 120);
}

void draw() {
  if (mousePressed) {
    fill(0);
  } else {
    fill(255);
  }
  ellipse(mouseX, mouseY, 80, 80);
}
```

This program creates a window 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 this program later. For now, run the code, move the mouse, and click to see what it does. While the sketch is running, the Run button will change to a square "stop" icon, which you can click to halt the sketch.

Show

If you don't want to use the buttons, you can always use the Sketch menu, which reveals the shortcut Ctrl-R (or Cmd-R on the Mac) for Run. The Present option clears the rest of the screen when the program is run to present the sketch all by itself. You can also use Present from the Toolbar by holding down the Shift key as you click the Run button. See Figure 2-2.

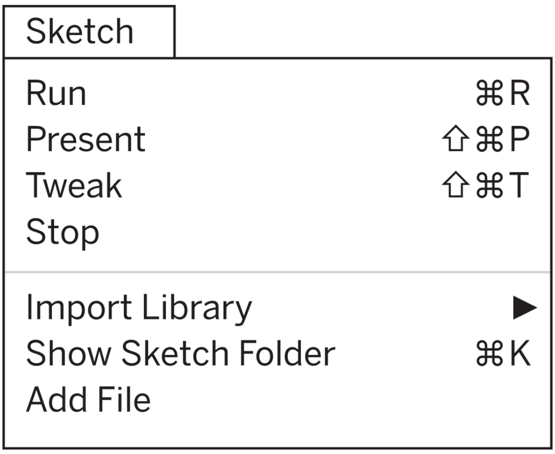


Figure 2-2. *A Processing sketch is displayed on screen with Run and Present. The Present option clears the entire screen before running the code for a cleaner presentation.*

Save and New

The next command that's important is Save. You can find it under the File menu. By default, your programs are saved to the "sketchbook," which is a folder that collects your programs for easy access. Select the Sketchbook option in the File menu to bring up a list of all the sketches in your sketchbook.

It's always a good idea to save your sketches often. As you try different things, keep saving with different names, so that you can always go back to an earlier version. This is especially helpful if—no, *when*—something breaks. You can also see where the sketch is located on your computer with the Show Sketch Folder command under the Sketch menu.

You can create a new sketch by selecting the New option from the File menu. This will create a new sketch in its own window.

Share

Processing sketches are made to be shared. The Export Application option in the File menu will bundle your code into a single folder. Export Application creates an application for your choice of Mac, Windows, and/or Linux. This is an easy way to make self-contained, double-clickable versions of your projects that can run full screen or in a window.

The application folders are erased and re-created each time you use the Export Application command, so be sure to move the folder elsewhere if you do not want it to be erased with the next export.

Examples and Reference

Learning how to program involves exploring lots of code: running, altering, breaking, and enhancing it until you have reshaped it into something new. With this in mind, the Processing software download includes dozens of examples that demonstrate different features of the software.

To open an *example*, select Examples from the File menu and double-click an example's name to open it. The examples are grouped into categories based on their function, such as Form, Motion, and Image. Find an interesting topic in the list and try an example.

All of the examples in this book can be downloaded and run from the Processing Development Environment. Open the examples through the File menu, then click Add Examples to open the list of example packages available to download. Select the *Getting Started with Processing* package and click Install to download.

When looking at code in the editor, you'll see that functions like `ellipse()` and `fill()` have a different color from the rest of the text. If you see a function that you're unfamiliar with, select the text, and then click "Find in Reference" from the Help menu. You can also right-click the text (or Ctrl-click on a Mac) and choose "Find in Reference" from the menu that appears. This will open a web browser and show the reference for that function. In addition, you can view the full documentation for the software by selecting Reference from the Help menu.

The *Processing 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 longer code found in the *Examples* folder. We recommend keeping the *Reference* open while you're reading this book and while you're programming. It can be navigated by topic or alphabetically; sometimes it's fastest to do a text search within your browser window.

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 over the years and reported them. If you think you can improve a reference entry or you find a mistake, please let us know by clicking the link at the top of each reference page.

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. In Processing, the *x* coordinate is the distance from the left edge of the Display Window and the *y* coordinate is the distance from the top edge. We write coordinates of a pixel like this: (x, y). So, if the screen 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 Display Window

The Display Window is created and images are drawn inside through code elements called *functions*. Functions are the basic building blocks of a Processing program. The behavior of a function is defined by its *parameters*. For example, almost every Processing program has a `size()` function to set the width and height of the Display Window. (If your program doesn't have a `size()` function, the dimension is set to 100×100 pixels.)

Example 3-1: Draw a Window

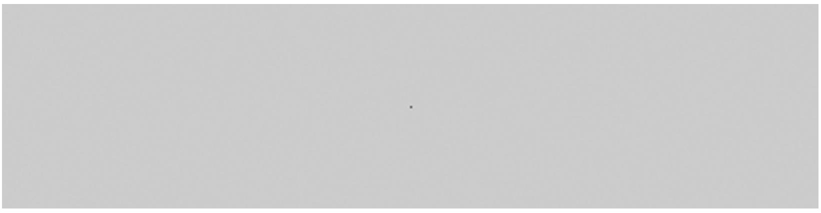
The `size()` function has two parameters: the first sets the width of the window and the second sets the height. To draw a window that is 800 pixels wide and 600 high, type:

```
size(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 Display Window, we use the `point()` function. It has two parameters that define a position: the *x* coordinate followed by the *y* coordinate. To draw a little window and a point at the center of the screen, coordinate (240, 60), type:



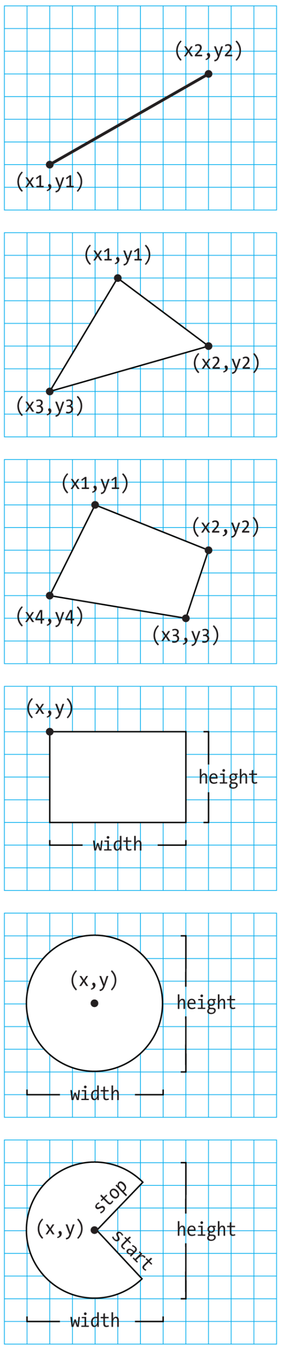
```
size(480, 120);
point(240, 60);
```

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

Basic Shapes

Processing 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.

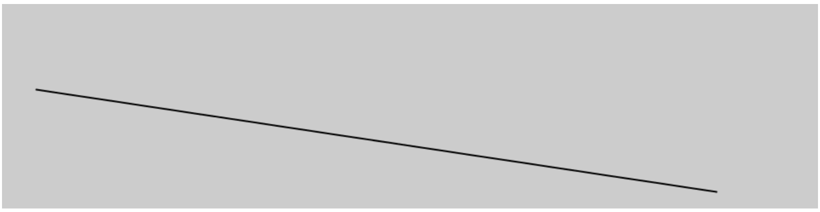


```
line(x1, y1, x2, y2)

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

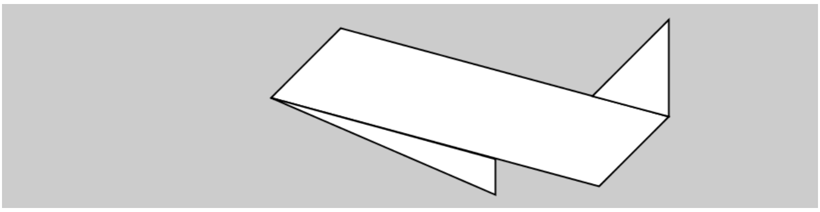
To draw a line between coordinate (20, 50) and (420, 110), try:



```
size(480, 120);
line(20, 50, 420, 110);
```

Example 3-4: Draw Basic Shapes

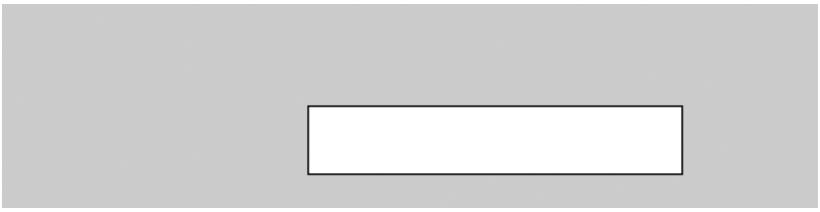
Following this pattern, a triangle needs six parameters and a quadrilateral needs eight (one pair for each point):



```
size(480, 120);
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

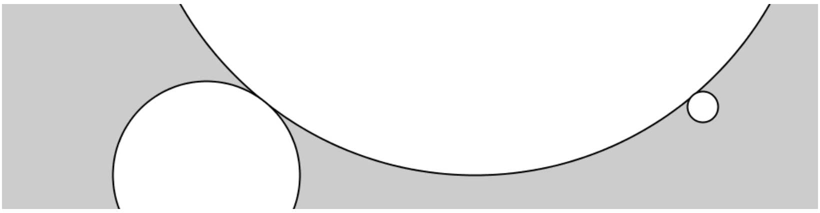
Rectangles and ellipses are both defined with four parameters: the first and second are for the *x* and *y* coordinates of the anchor point, the third for the width, and the fourth for the height. To make a rectangle at coordinate (180, 60) with a width of 220 pixels and height of 40, use the `rect()`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



```
size(480, 120);
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 window. Objects can be drawn partially (or entirely) out of the window without an error:

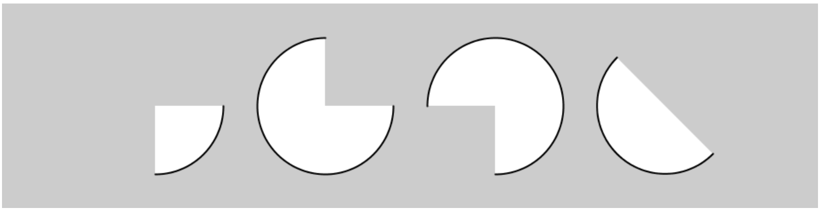


```
size(480, 120);
ellipse(278, -100, 400, 400);
ellipse(120, 100, 110, 110);
ellipse(412, 60, 18, 18);
```

Processing 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 to `ellipse()` and `rect()`.

Example 3-7: Draw Part of an Ellipse

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



```
size(480, 120);
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, 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 Processing. The values `PI`, `QUARTER_PI`, `HALF_PI`, and `TWO_PI` can be used to replace the radian values for 180°, 45°, 90°, and 360°.

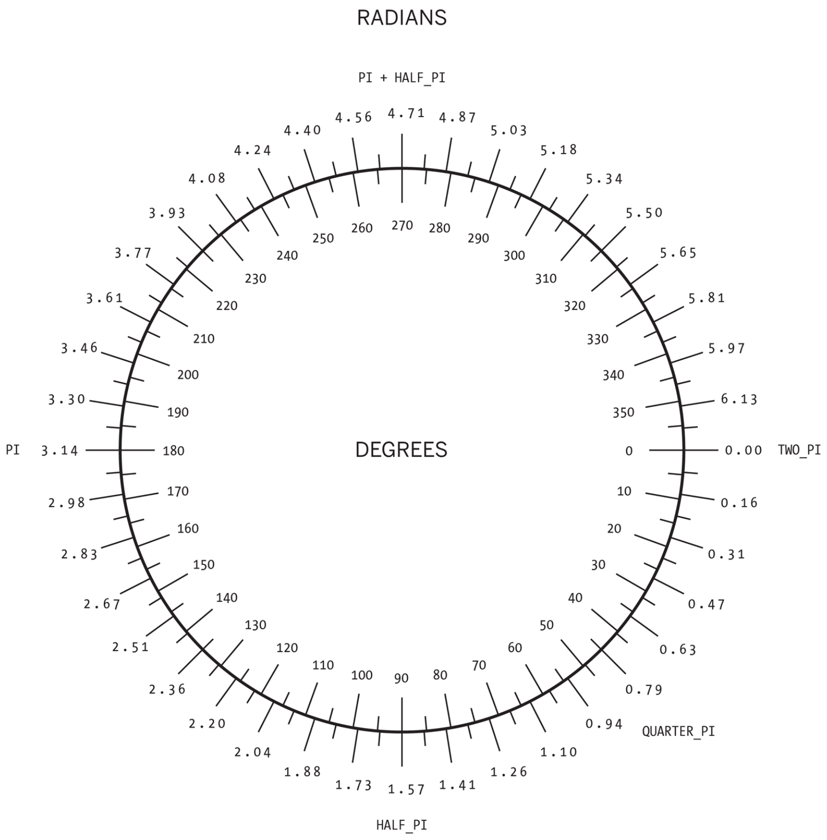


Figure 3-2. *Radians and degrees are two ways to measure an angle. Degrees move around the circle from 0 to 360, while radians measure the angles in relation to pi, from 0 to approximately 6.28.*

Example 3-8: Draw with Degrees

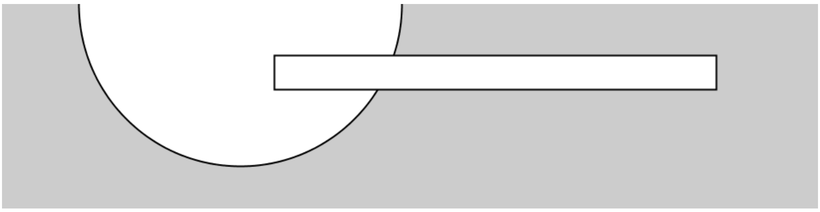
If you prefer to use degree measurements, you can convert to radians with the `radians()` function. This function takes an angle in degrees and changes it to the corresponding radian value. The following example is the same as Example 3-7 on page 18, but it uses the `radians()` function to define the start and stop values in degrees:

```
size(480, 120);
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));
```

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

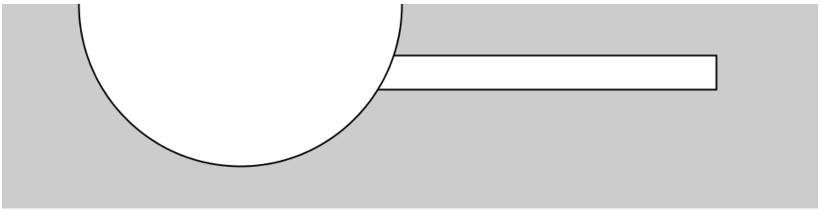
Example 3-9: Control Your Drawing Order



```
size(480, 120);
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 3-10: Put It in Reverse

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



```
size(480, 120);
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);
```

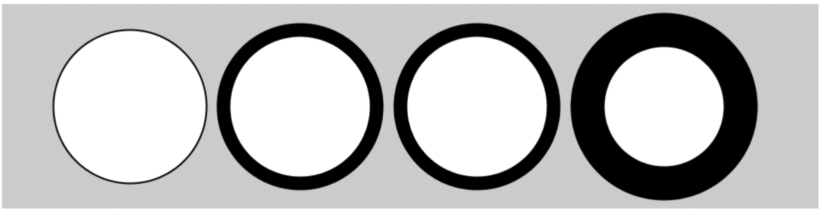
You can think of it like painting with a brush or making a collage. The last element that you add is what's visible on top.

Shape Properties

The most basic and useful shape properties are stroke weight, the way the ends (caps) of lines are drawn, and how the corners of shapes are displayed.

Example 3-11: Set Stroke Weight

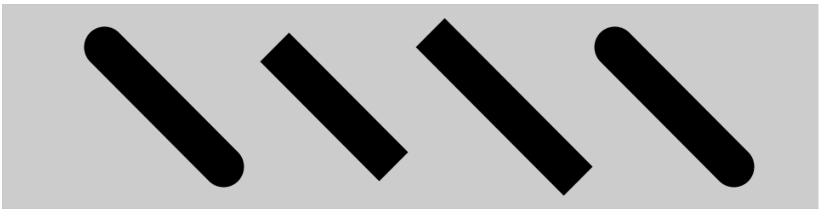
The default stroke weight is a single pixel, but this can be changed with the `strokeWeight()` function. The single parameter to `strokeWeight()` sets the width of drawn lines:



```
    size(480, 120);
    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 3-12: Set Stroke Caps

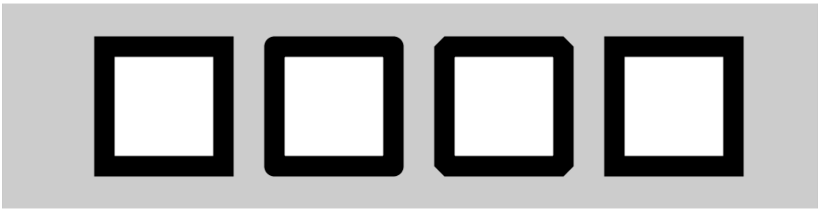
The `strokeCap()` function changes how lines are drawn at their endpoints. By default, they have rounded ends:



```
size(480, 120);
strokeWeight(24);
line(60, 25, 130, 95);
strokeCap(SQUARE);   // Square the line endings
line(160, 25, 230, 95);
strokeCap(PROJECT);  // Project the line endings
line(260, 25, 330, 95);
strokeCap(ROUND);    // Round the line endings
line(360, 25, 430, 95);
```

Example 3-13: Set Stroke Joins

The `strokeJoin()` function changes the way lines are joined (how the corners look). By default, they have pointed (mitered) corners:



```
size(480, 120);
strokeWeight(12);
rect(60, 25, 70, 70);
strokeJoin(ROUND);   // Round the stroke corners
rect(160, 25, 70, 70);
strokeJoin(BEVEL);   // Bevel the stroke corners
rect(260, 25, 70, 70);
strokeJoin(MITER);   // Miter the stroke corners
rect(360, 25, 70, 70);
```

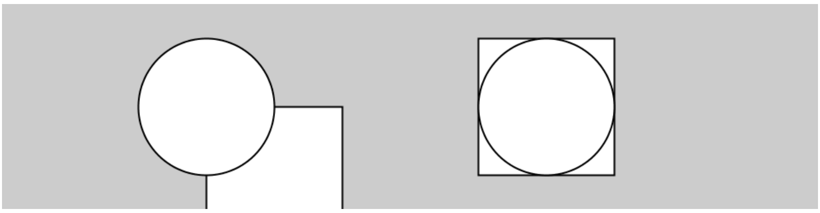
When any of these attributes are set, all shapes drawn afterward are affected. For instance, in Example 3-11 on page 21, notice how the second and third circles both have the same stroke weight, even though the weight is set only once before both are drawn.

Drawing Modes

A group of functions with "mode" in their name change how Processing draws geometry to the screen. In this chapter, we'll look at `ellipseMode()` and `rectMode()`, which help us to draw ellipses and rectangles, respectively; later in the book, we'll cover `imageMode()` and `shapeMode()`.

Example 3-14: On the Corner

By default, the `ellipse()` function uses its first two parameters as the *x* and *y* coordinate of the center and the third and fourth parameters as the width and height. After `ellipseMode(CORNER)` is run in a sketch, the first two parameters to `ellipse()` then define the position of the upper-left corner of the rectangle the ellipse is inscribed within. This makes the `ellipse()` function behave more like `rect()` as seen in this example:

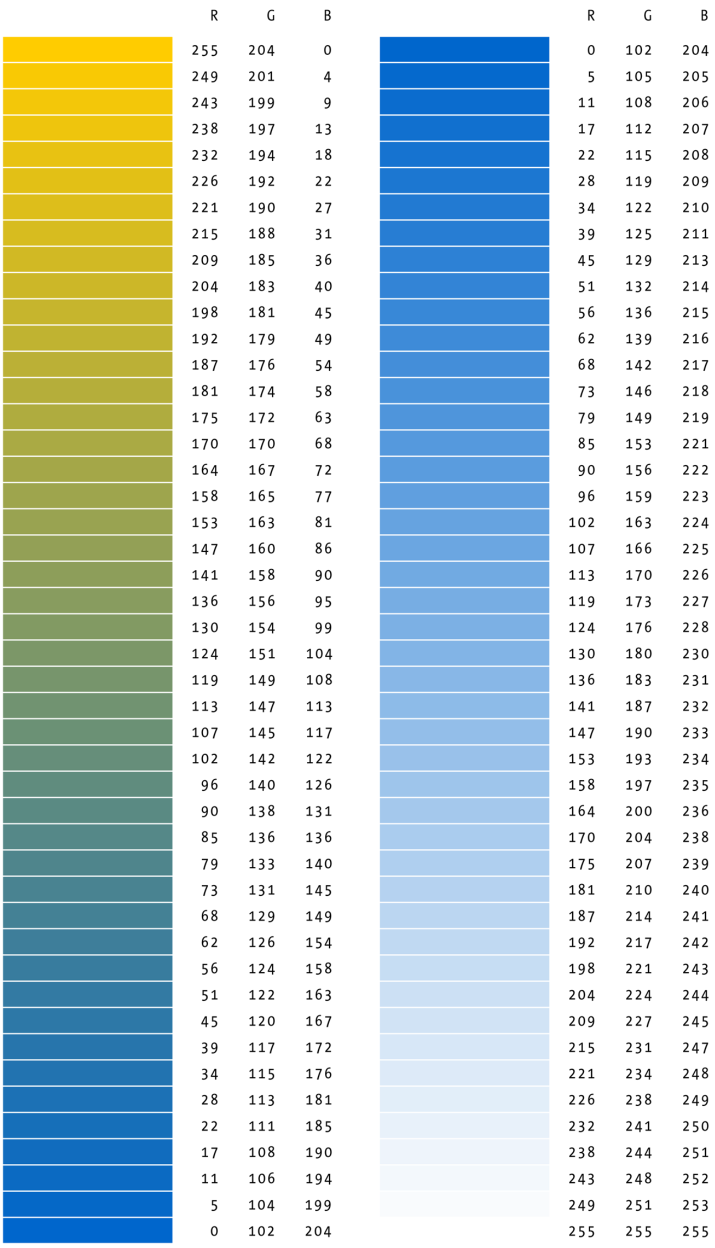


```
size(480, 120);
rect(120, 60, 80, 80);
ellipse(120, 60, 80, 80);
ellipseMode(CORNER);
rect(280, 20, 80, 80);
ellipse(280, 20, 80, 80);
```

You'll find these "mode" functions in examples throughout the book. There are more options for how to use them in the *Processing Reference*.

Color

All the shapes so far have been filled white with black outlines, and the background of the Display Window has been light gray. To change them, use the `background()`, `fill()`, and `stroke()` functions. The values of the parameters are in the range of 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.



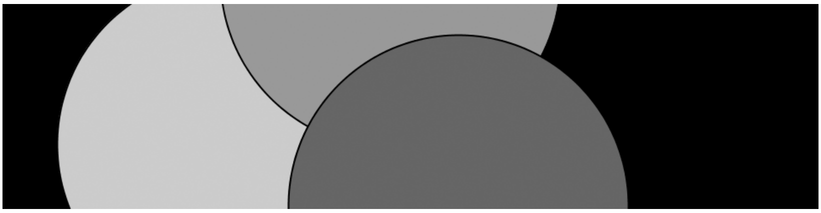
R	G	B
255	204	0
249	201	4
243	199	9
238	197	13
232	194	18
226	192	22
221	190	27
215	188	31
209	185	36
204	183	40
198	181	45
192	179	49
187	176	54
181	174	58
175	172	63
170	170	68
164	167	72
158	165	77
153	163	81
147	160	86
141	158	90
136	156	95
130	154	99
124	151	104
119	149	108
113	147	113
107	145	117
102	142	122
96	140	126
90	138	131
85	136	136
79	133	140
73	131	145
68	129	149
62	126	154
56	124	158
51	122	163
45	120	167
39	117	172
34	115	176
28	113	181
22	111	185
17	108	190
11	106	194
5	104	199
0	102	204

R	G	B
0	102	204
5	105	205
11	108	206
17	112	207
22	115	208
28	119	209
34	122	210
39	125	211
45	129	213
51	132	214
56	136	215
62	139	216
68	142	217
73	146	218
79	149	219
85	153	221
90	156	222
96	159	223
102	163	224
107	166	225
113	170	226
119	173	227
124	176	228
130	180	230
136	183	231
141	187	232
147	190	233
153	193	234
158	197	235
164	200	236
170	204	238
175	207	239
181	210	240
187	214	241
192	217	242
198	221	243
204	224	244
209	227	245
215	231	247
221	234	248
226	238	249
232	241	250
238	244	251
243	248	252
249	251	253
255	255	255

Figure 3-3. *Colors are created by defining RGB (red, green, blue) values*

Example 3-15: Paint with Grays

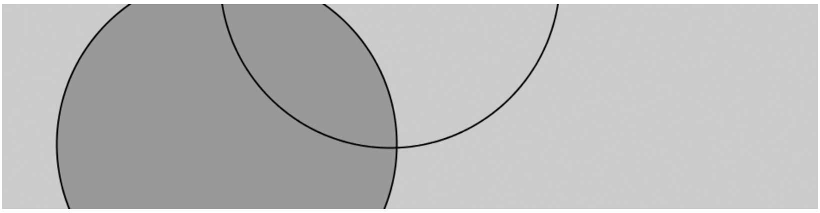
This example shows three different gray values on a black background:



```
size(480, 120);
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 3-16: Control Fill and Stroke

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



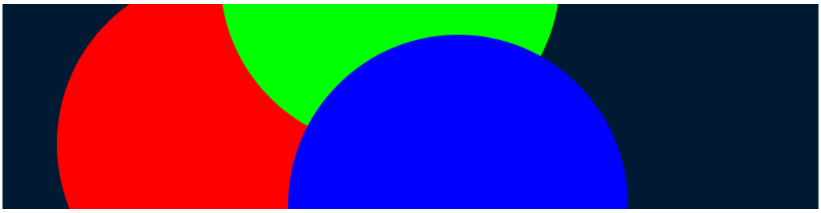
```
size(480, 120);
fill(153);                     // Medium gray
ellipse(132, 82, 200, 200);    // Gray circle
noFill();                      // Turn off fill
ellipse(228, -16, 200, 200);   // Outline circle
noStroke();                    // Turn off stroke
ellipse(268, 118, 200, 200);   // Doesn't 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 3-17: Draw with Color

To move beyond grayscale values, you use three parameters to specify the red, green, and blue components of a color.

Run the code in Processing to reveal the colors:



```
size(480, 120);
noStroke();
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
```

This is referred to as RGB color, which comes from 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. Using RGB color isn't very intuitive, so to choose colors, use Tools→Color Selector, which shows a color palette similar to those found in other software (see Figure 3-4). Select a color, and then use the R, G, and B values as the parameters for your `background()`, `fill()`, or `stroke()` function.

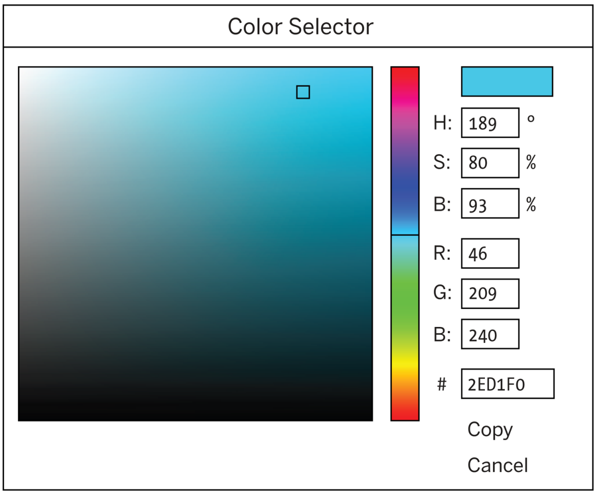
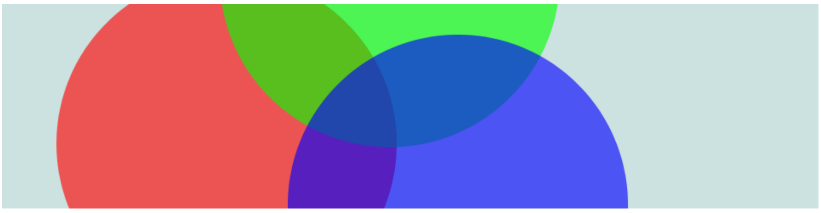


Figure 3-4. *Processing Color Selector*

Example 3-18: 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:



```
size(480, 120);
noStroke();
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
```

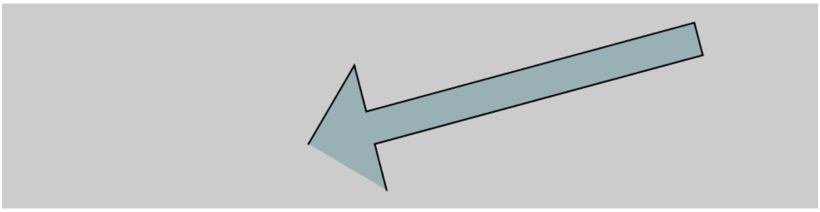
```
fill(0, 0, 255, 160);         // Blue color
ellipse(268, 118, 200, 200);  // Blue circle
```

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 3-19: Draw an Arrow

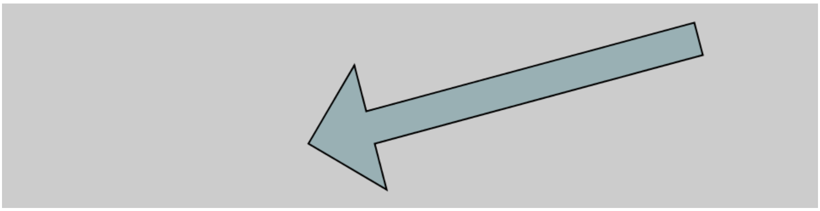
The `beginShape()` function signals the start of a new shape. The `vertex()` function is used to define each pair of *x* and *y* coordinates for the shape. Finally, `endShape()` is called to signal that the shape is finished:



```
size(480, 120);
beginShape();
fill(153, 176, 180);
vertex(180, 82);
vertex(207, 36);
vertex(214, 63);
vertex(407, 11);
vertex(412, 30);
vertex(219, 82);
vertex(226, 109);
endShape();
```

Example 3-20: Close the Gap

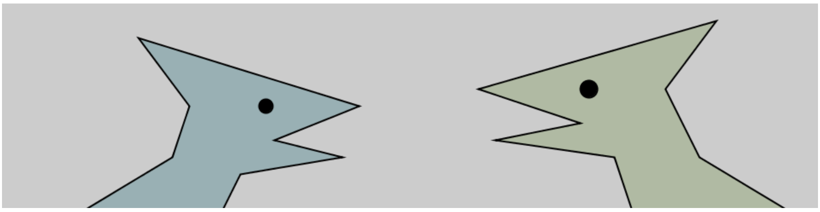
When you run Example 3-19 on page 28, you’ll see the first and last point are not connected. To do this, add the word `CLOSE` as a parameter to `endShape()`, like this:



```
size(480, 120);
beginShape();
fill(153, 176, 180);
vertex(180, 82);
vertex(207, 36);
vertex(214, 63);
vertex(407, 11);
vertex(412, 30);
vertex(219, 82);
vertex(226, 109);
endShape(CLOSE);
```

Example 3-21: Create Some Creatures

The power of defining shapes with `vertex()` is the ability to make shapes with complex outlines. Processing 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:



```
size(480, 120);

// Left creature
fill(153, 176, 180);
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(176, 186, 163);
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 when trying to choose the right color. So, for instance, I might be trying to find just the right red for an ellipse:

```
size(200, 200);
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:

```
size(200, 200);
//fill(165, 57, 57);
fill(144, 39, 39);
ellipse(100, 100, 80, 80);
```

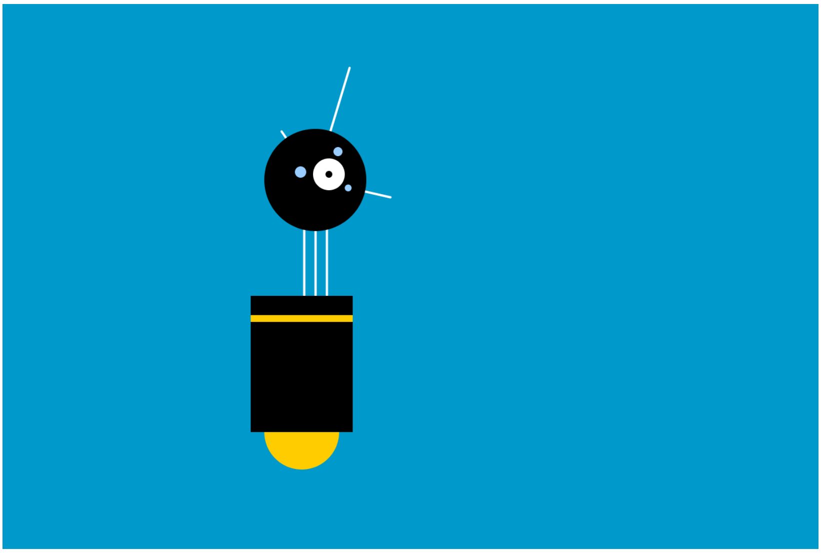
Placing // at the beginning of the line temporarily disables it. Or I can remove the // and place it in front of the other line if I want to try it again:

```
size(200, 200);
fill(165, 57, 57);
//fill(144, 39, 39);
ellipse(100, 100, 80, 80);
```

As you work with Processing 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.

As a shortcut, you can also use Ctrl-/ (Cmd-/ on the Mac) to add or remove comments from the current line or a selected block of text. You can also comment out many lines at a time with the alternative comment notation introduced in "Comments" on page 203.

Robot 1: Draw



This is P5, the Processing Robot. There are 10 different programs to draw and animate him in the 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 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:

```
size(720, 480);
strokeWeight(2);
background(0, 153, 204);    // Blue background
ellipseMode(RADIUS);

// Neck
stroke(255);                // Set stroke to white
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(255, 204, 0);          // Set fill to orange
ellipse(264, 377, 33, 33);  // Antigravity orb
fill(0);                    // Set fill to black
rect(219, 257, 90, 120);    // Main body
fill(255, 204, 0);          // Set fill to yellow
rect(219, 274, 90, 6);      // Yellow 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, 204, 255);        // Set fill to light blue
ellipse(263, 148, 5, 5);    // Small eye 1
ellipse(296, 130, 4, 4);    // Small eye 2
ellipse(305, 162, 3, 3);    // Small eye 3
```

4/Variables

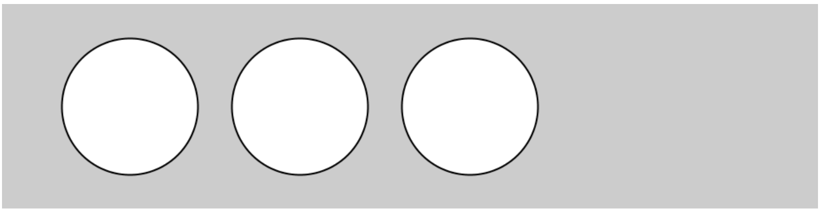
A *variable* stores a value in memory so that it can be used later in a program. The variable can be used many times within a single program, and the value is easily changed while the program is running.

First Variables

One of the reasons 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

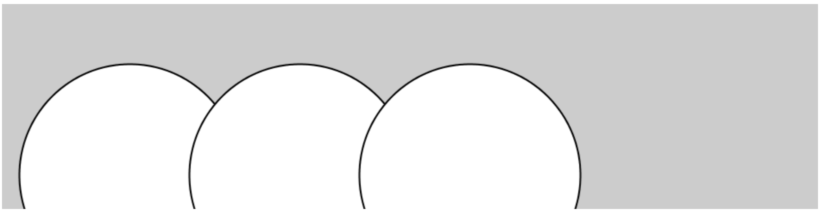
For instance, when you make the *y* coordinate and diameter for the three circles in this example into variables, the same values are used for each ellipse:



```
size(480, 120);
int y = 60;
int d = 80;
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 alters all three ellipses:



```
size(480, 120);
int y = 100;
int d = 130;
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 Example 4-1 on page 35 and Example 4-2 on page 36, notice how the bottom three lines are the same, and only the middle 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*, the *data type*, 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.

The range of values that can be stored within a variable is defined by its *data type*. For instance, the *integer* data type can store numbers without decimal places (whole numbers). In code, *integer* is abbreviated to `int`. There are data types to store each kind of data: integers, floating-point (decimal) numbers, characters, words, images, fonts, and so on.

Variables must first be *declared*, which sets aside space in the computer's memory to store the information. When declaring a variable, you also need to specify its data type (such as `int`), which indicates what kind of information is being stored. After the data type and name are set, a value can be assigned to the variable:

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

This code does the same thing, but is shorter:

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

The name of the data type is included on the line of code that declares a variable, but it's not written again. Each time the data type 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 D), so the program has an error:

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

Each data type stores a different kind of data. For instance, an `int` variable can store a whole number, but it can't store a number with decimal points, called a `float`. The word "`float`" refers to "floating point," which describes the technique used to store a number with decimal points in memory. (The specifics of that technique aren't important here.)

A floating-point number can't be assigned to an `int` because information would be lost. For instance, the value 12.2 is different from its nearest `int` equivalent, the value 12. In code, this operation will create an error:

```
int x = 12.2;  // ERROR! A floating-point value can't fit in
an int
```

However, a `float` variable can store an integer. For instance, the integer value 12 can be converted to the floating-point equivalent 12.0 because no information is lost. This code works without an error:

```
float x = 12;  // Automatically converts 12 to 12.0
```

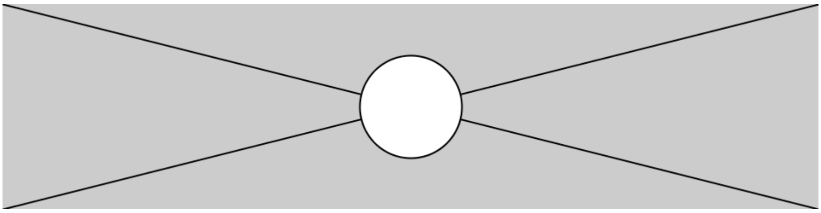
Data types are discussed in more detail in Appendix B.

Processing Variables

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

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

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



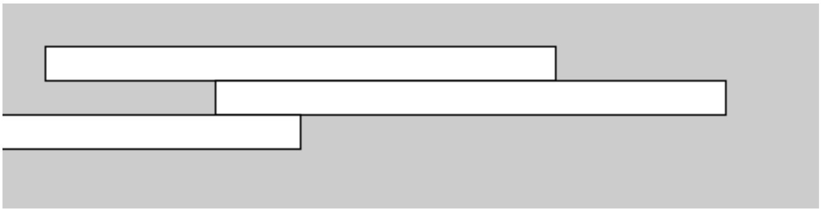
```
size(480, 120);
line(0, 0, width, height); // Line from (0,0) to (480, 120)
line(width, 0, 0, height); // Line from (480, 0) to (0, 120)
ellipse(width/2, height/2, 60, 60);
```

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



```
size(480, 120);
int x = 25;
int h = 20;
int y = 25;
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

Processing 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:

```
int 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:

```
int 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:

```
int 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 C.

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
```

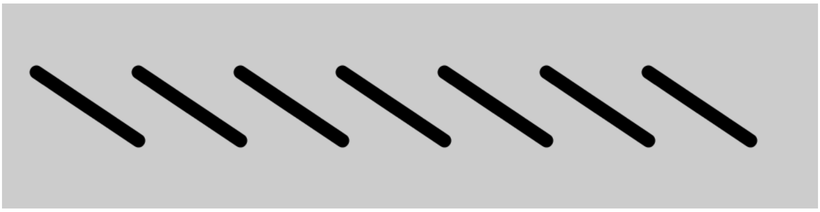
More shortcuts can be found in the *Processing Reference*.

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:



```
size(480, 120);
strokeWeight(8);
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:

```
size(480, 120);
strokeWeight(8);
for (int 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` sets the starting value, often declaring a new variable to use within the `for` loop. In the earlier example, an integer named `i` was declared and set to 20. The variable name `i` is frequently used, but there's really nothing special about it. The *test* evaluates the value of this variable (here, it checks whether `i` still less than 400), and the *update* changes the variable's value (adding 60 before repeating the loop). Figure 4-1 shows the order in which they run and how they control the code statements inside the block.

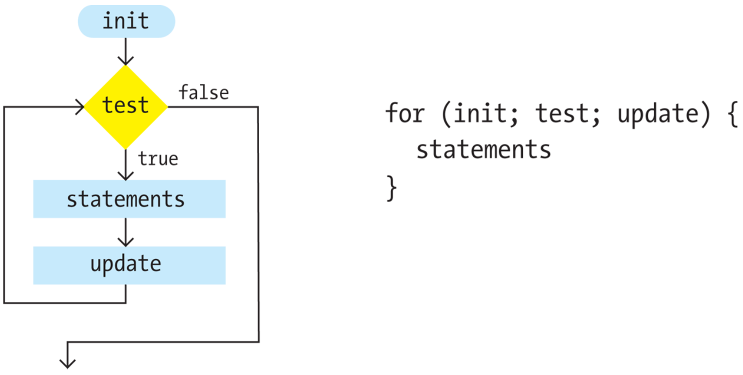


Figure 4-1. *Flow diagram of a for loop*

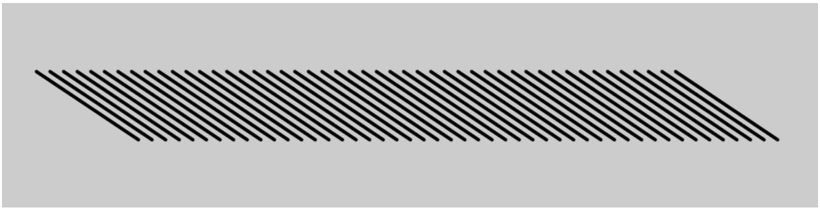
The *test* statement requires more explanation. It's always a *relational expression* that compares two values with a *relational operator*. In this example, the expression is "i < 400" and the operator is the `<` (less than) symbol. The most common relational operators are:

>	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.

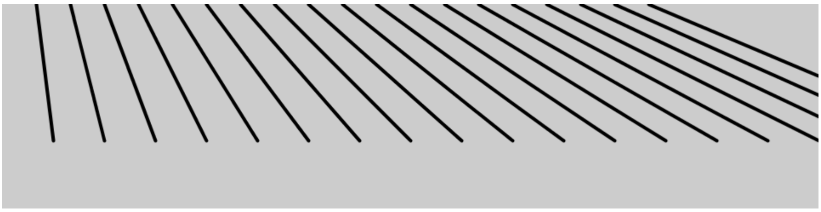
Example 4-7: Flex Your for Loop’s Muscles

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 on page 41 only slightly, we can create a range of different patterns:



```
size(480, 120);
strokeWeight(2);
for (int i = 20; i < 400; i += 8) {
  line(i, 40, i + 60, 80);
}
```

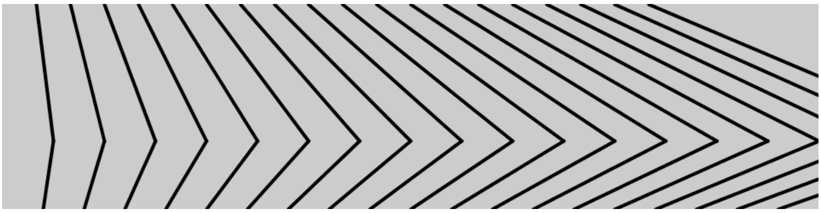
Example 4-8: Fanning Out the Lines



```
size(480, 120);
strokeWeight(2);
```

```
for (int i = 20; i < 400; i += 20) {
  line(i, 0, i + i/2, 80);
}
```

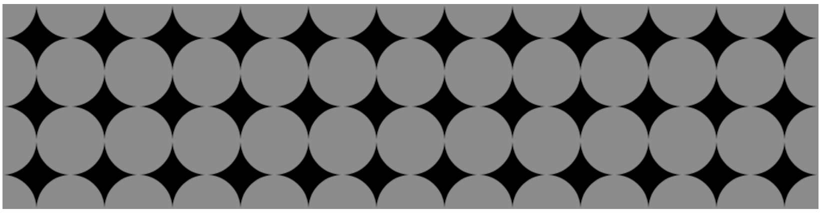
Example 4-9: Kinking the Lines



```
size(480, 120);
strokeWeight(2);
for (int 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

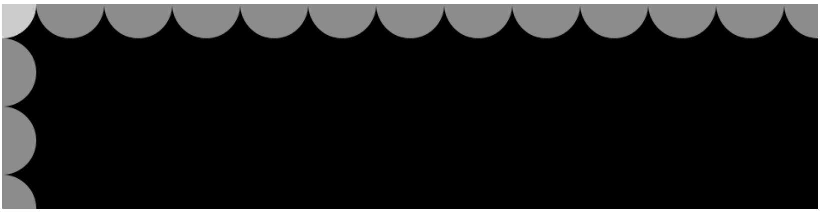
When one `for` loop is embedded inside another, the number of repetitions is multiplied. First, let's look at a short example, and then we'll break it down in Example 4-11 on page 45:



```
size(480, 120);
background(0);
noStroke();
for (int y = 0; y <= height; y += 40) {
  for (int 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:

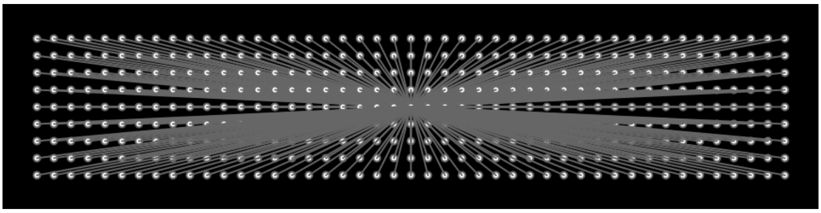


```
size(480, 120);
background(0);
noStroke();
for (int y = 0; y < height+45; y += 40) {
  fill(255, 140);
  ellipse(0, y, 40, 40);
}
for (int x = 0; x < width+45; x += 40) {
  fill(255, 140);
  ellipse(x, 0, 40, 40);
}
```

When one of these `for` loops is placed inside the other, as in Example 4-10 on page 44, 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).

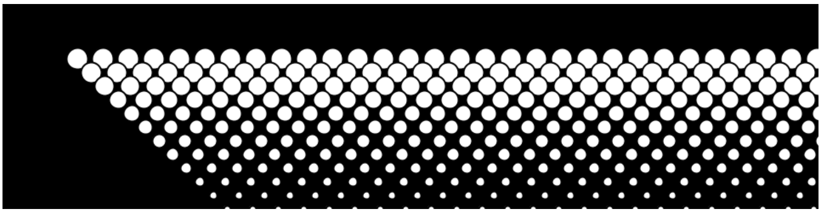
Example 4-10 on page 44 is a good base for exploring many types of repeating visual patterns. The following examples show a couple of ways that it can be extended, but this is only a tiny sample of what's possible. In Example 4-12 on page 46, the code draws a line from each point in the grid to the center of the screen. In Example 4-13 on page 46, the ellipses shrink with each new row and are moved to the right by adding the *y* coordinate to the *x* coordinate.

Example 4-12: Pins and Lines



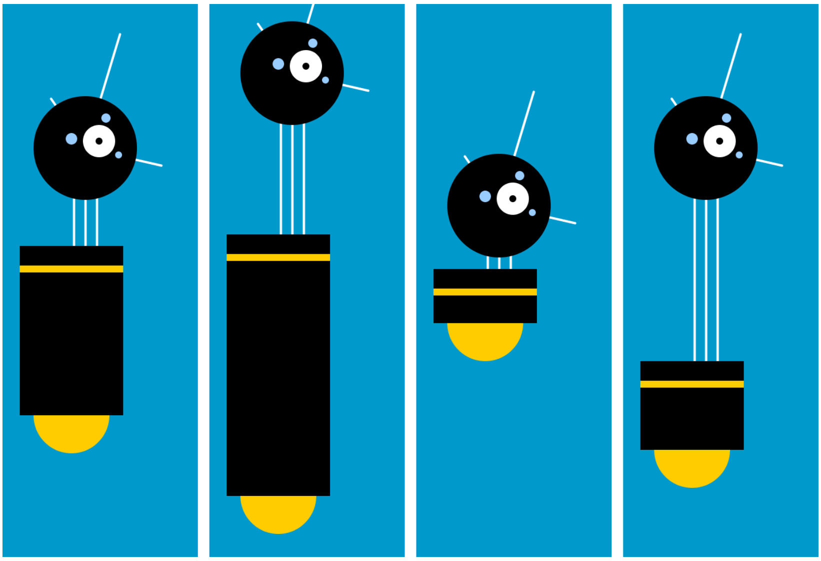
```
size(480, 120);
background(0);
fill(255);
stroke(102);
for (int y = 20; y <= height-20; y += 10) {
  for (int x = 20; x <= width-20; x += 10) {
    ellipse(x, y, 4, 4);
    // Draw a line to the center of the display
    line(x, y, 240, 60);
  }
}
```

Example 4-13: Halftone Dots



```
size(480, 120);
background(0);
for (int y = 32; y <= height; y += 8) {
  for (int 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" on page 32), but now it's much easier to modify, because numbers that depend on one another are in a single location. For instance, the neck can be drawn based on the `bodyHeight` 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:

```
int x = 60;              // x coordinate
int y = 390;             // y coordinate
int bodyHeight = 180;    // Body height
int neckHeight = 40;     // Neck height
int radius = 45;
int ny = y - bodyHeight - neckHeight - radius;  // Neck y

size(170, 480);
strokeWeight(2);
background(0, 153, 204);
ellipseMode(RADIUS);

// Neck
stroke(255);
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(255, 204, 0);
ellipse(x, y-33, 33, 33);
fill(0);
rect(x-45, y-bodyHeight, 90, bodyHeight-33);
fill(255, 204, 0);
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, 204, 255);
ellipse(x, ny-8, 5, 5);
ellipse(x+30, ny-26, 4, 4);
ellipse(x+41, ny+6, 3, 3);
```