Lesson One

The Beginning

This page intentionally left blank

1 Pixels

A journey of a thousand miles begins with a single step.
—Lao-tzu

In this chapter:
- Specifying pixel coordinates
- Basic shapes: point, line, rectangle, ellipse
- Color: grayscale, RGB
- Color: alpha transparency

Note that you are not doing any programming yet in this chapter! You are just dipping your feet in the water and getting comfortable with the idea of creating onscreen graphics with text-based commands, that is, "code"!

1-1 Graph paper

This book will teach you how to program in the context of computational media, and it will use the development environment Processing (*http://www.processing.org*) as the basis for all discussion and examples. But before any of this becomes relevant or interesting, you must first channel your eighth-grade self, pull out a piece of graph paper, and draw a line. The shortest distance between two points is a good old fashioned line, and this is where you will begin, with two points on that graph paper.

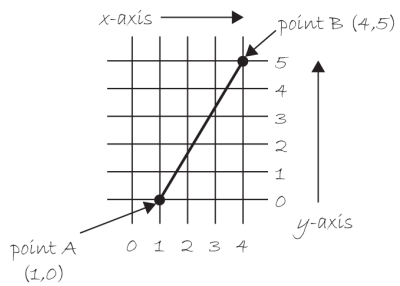


Figure 1-1

Figure 1-1 shows a line between point A (1,0) and point B (4,5). If you wanted to direct a friend of yours to draw that same line, you would say "draw a line from the point one-zero to the point four-five, please." Well, for the moment, imagine your friend was a computer and you wanted to instruct this digital pal to display that same line on its screen. The same command applies (only this time you can skip the pleasantries and you will be required to employ a precise formatting). Here, the instruction will look like this:

```
line(1, 0, 4, 5);
```

Congratulations, you have written your first line of computer code! I'll will get to the precise formatting of the above later, but for now, even without knowing too much, it should make a fair amount of sense. I am providing a *command* (which I will refer to as a *function*) named *line* for the machine to follow. In addition, I am specifying some *arguments* for how that line should be drawn, from point A (1,0) to point B (4,5). If you think of that line of code as a sentence, the *function* is a *verb* and the *arguments* are the *objects* of the sentence. The code sentence also ends with a semicolon instead of a period.

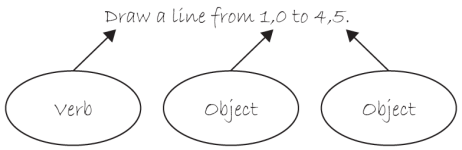


Figure 1-2

The key here is to realize that the computer screen is nothing more than a *fancier* piece of graph paper. Each pixel of the screen is a coordinate — two numbers, an *x* (horizontal) and a *y* (vertical) — that determine the location of a point in space. And it's your job to specify what shapes and colors should appear at these pixel coordinates.

Nevertheless, there is a catch here. The graph paper from eighth grade (*Cartesian coordinate system*) placed (0,0) in the center with the y-axis pointing up and the x-axis pointing to the right (in the positive direction, negative down and to the left). The coordinate system for pixels in a computer window, however, is reversed along the y-axis. (0,0) can be found at the top left with the positive direction to the right horizontally and down vertically. See Figure 1-3.

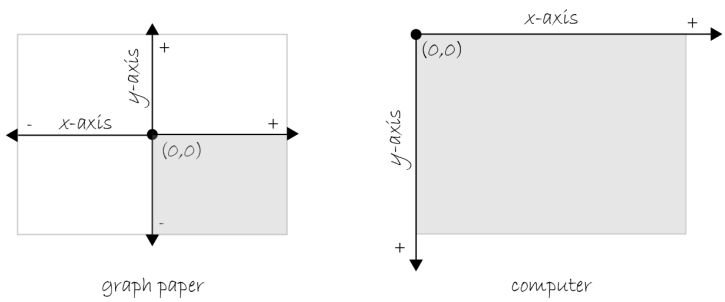


Figure 1-3

Exercise 1-1: Looking at how I wrote the instruction for line — `line(1, 0, 4, 5);` — how would you guess you would write an instruction to draw a rectangle? A circle? A triangle? Write out the instructions in English and then translate it into code.

English: _____

Code: _____

English: _____

Code: _____

English: _____

Code: _____

Come back later and see how your guesses matched up with how Processing actually works.

1-2 Simple shapes

The vast majority of the programming examples in this book will be visual in nature. You may ultimately learn to develop interactive games, algorithmic art pieces, animated logo designs, and (insert your own category here) with Processing, but at its core, each visual program will involve setting pixels. The simplest way to get started in understanding how this works is to learn to draw primitive shapes. This is not unlike how you learn to draw in elementary school, only here you do so with code instead of crayons.

I'll start with the four primitive shapes shown in Figure 1-4.

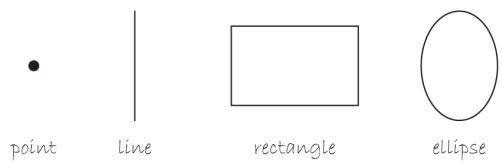


Figure 1-4

For each shape, ask yourself what information is required to specify the location and size (and later color) of that shape and learn how Processing expects to receive that information. In each of the diagrams below (Figure 1-5 through Figure 1-11), assume a window with a width of ten pixels and height of ten pixels. This isn't particularly realistic since when you really start coding you will most likely work with much larger windows (ten by ten pixels is barely a few millimeters of screen space). Nevertheless, for demonstration purposes, it's nice to work with smaller numbers in order to present the pixels as they might appear on graph paper (for now) to better illustrate the inner workings of each line of code.

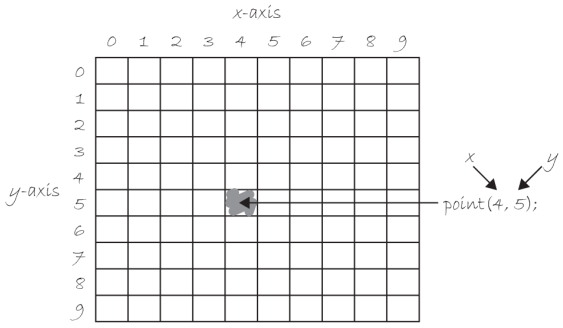


Figure 1-5

A point is the easiest of the shapes and a good place to start. To draw a point, you only need an (x,y) coordinate as shown in Figure 1-5. A line isn't terribly difficult either. A line requires two points, as shown in Figure 1-6.

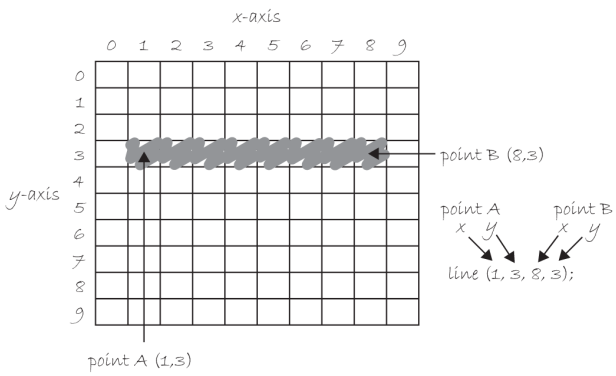


Figure 1-6

Once you arrive at drawing a rectangle, things become a bit more complicated. In Processing, a rectangle is specified by the coordinate for the top left corner of the rectangle, as well as its width and height (see Figure 1-7).

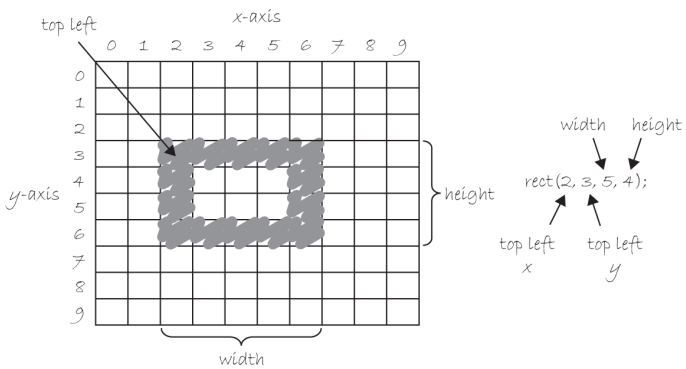


Figure 1-7

However, a second way to draw a rectangle involves specifying the centerpoint, along with width and height as shown in Figure 1-8. If you prefer this method, you first indicate that you want to use the

CENTER mode before the instruction for the rectangle itself. Note that Processing is case-sensitive. Incidentally, the default mode is CORNER, which is how I began as illustrated in Figure 1-7.

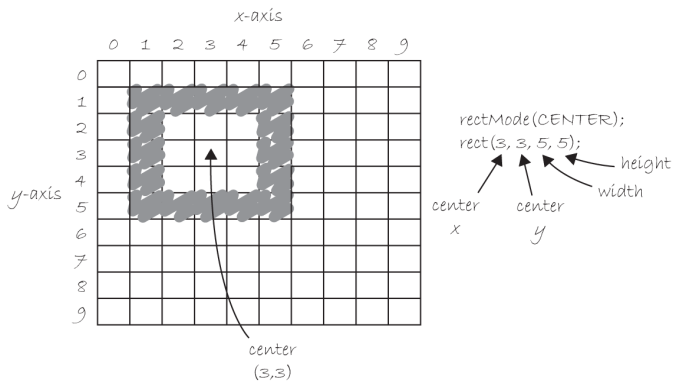


Figure 1-8

Finally, you can also draw a rectangle with two points (the top left corner and the bottom right corner). The mode here is CORNERS (see Figure 1-9).

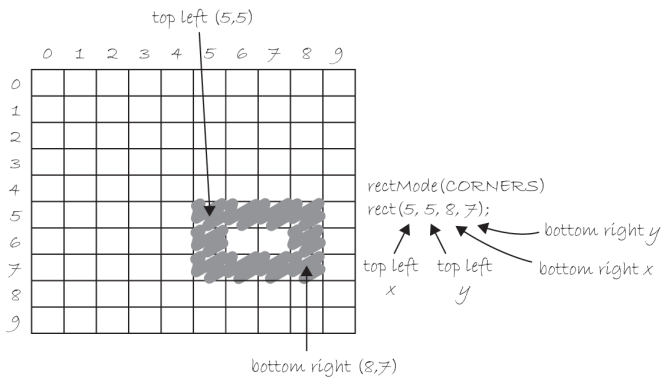


Figure 1-9

Once you have become comfortable with the concept of drawing a rectangle, an ellipse is a snap. In fact, it's identical to `rect()` with the difference being that an ellipse is drawn where the bounding box[1] (as shown in Figure 1-10) of the rectangle would be. The default mode for `ellipse()` is CENTER, rather than CORNER as with `rect()`. See Figure 1-11.

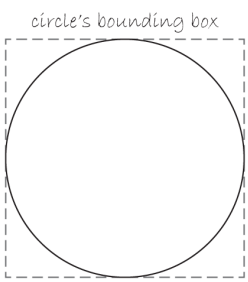


Figure 1-10

[1] A bounding box of a shape in computer graphics is the smallest rectangle that includes all the pixels of that shape. For example, the bounding box of a circle is shown in Figure 1-10.

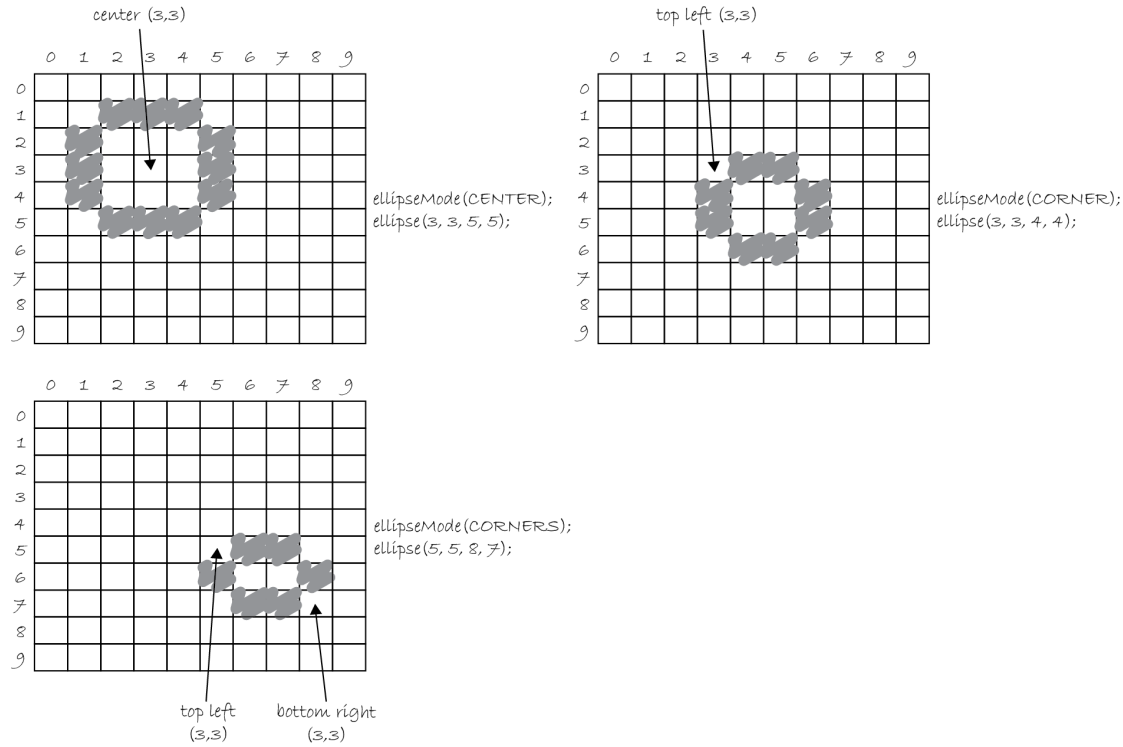


Figure 1-11

It's important to acknowledge that in Figure 1-11, the ellipses do not look particularly circular. Processing has a built-in methodology for selecting which pixels should be used to create a circular shape. Zoomed in like this, you get a bunch of squares in a circle-like pattern, but zoomed out on a computer screen, you get a nice round ellipse. Later, you will see that Processing gives you the power to develop your own algorithms for coloring in individual pixels (in fact, you can probably already imagine how you might do this using `point()` over and over again), but for now, it's best to let `ellipse()` do the hard work.

Certainly, point, line, ellipse, and rectangle are not the only shapes available in the Processing library of functions. In Chapter 2, you will see how the Processing reference provides a full list of available drawing functions along with documentation of the required arguments, sample syntax, and imagery. For now, as an exercise, you might try to imagine what arguments are required for some other shapes (Figure 1-12): `triangle()`, `arc()`, `quad()`, `curve()`.

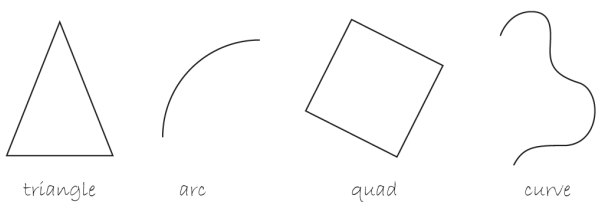
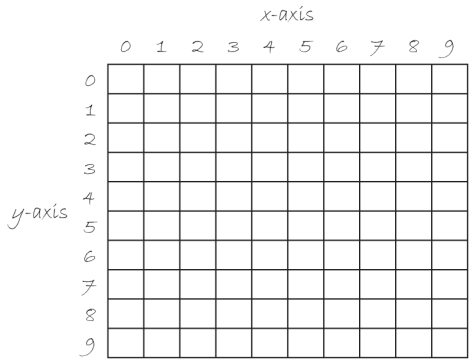


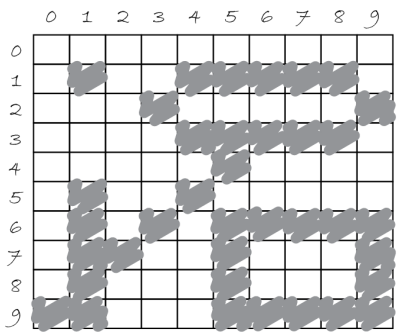
Figure 1-12

Exercise 1-2: Using the blank graph below, draw the primitive shapes specified by the code.

```
line(0, 0, 9, 6);
point(0, 2);
point(0, 4);
rectMode(CORNER);
rect(5, 0, 4, 3);
ellipseMode(CENTER);
ellipse(3, 7, 4, 4);
```



Exercise 1-3: Reverse engineer a list of primitive shape drawing instructions for the diagram below.



Note: There is more than one correct answer!

1-3 Grayscale color

As you learned in Section 1-2 on page 5, the primary building block for placing shapes onscreen is a pixel coordinate. You politely instructed the computer to draw a shape at a specific location with a specific size. Nevertheless, a fundamental element was missing — color.

In the digital world, precision is required. Saying "Hey, can you make that circle bluish-green?" will not do. Therefore, color is defined with a range of numbers. I'll start with the simplest case: *black and white* or *grayscale*. To specify a value for grayscale, use the following: 0 means black, 255 means white. In between, every other number — 50, 87, 162, 209, and so on — is a shade of gray ranging from black to white. See Figure 1-13.

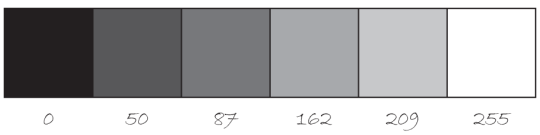


Figure 1-13

Does 0–255 seem arbitrary to you?

Color for a given shape needs to be stored in the computer's memory. This memory is just a long sequence of 0's and 1's (a whole bunch of on or off switches.) Each one of these switches is a *bit*, eight of them together is a *byte*. Imagine if you had eight bits (one byte) in sequence — how many ways can you configure these switches? The answer is (and doing a little research into binary numbers will prove this point) 256 possibilities, or a range of numbers between 0 and 255. Processing will use eight bit color for the grayscale range and 24 bit for full color (eight bits for each of the red, green, and blue color components; see Section 1-4 on page 12).

Understanding how this range works, you can now move to setting specific grayscale colors for the shapes you drew in Section 1-2 on page 5. In Processing, every shape has a `stroke()` or a `fill()` or both. The `stroke()` specifies the color for the outline of the shape, and the `fill()` specifies the color for the interior of that shape. Lines and points can only have `stroke()`, for obvious reasons.

If you forget to specify a color, Processing will use black (0) for the `stroke()` and white (255) for the `fill()` by default. Note that I'm now using more realistic numbers for the pixel locations, assuming a larger window of size 200 × 200 pixels. See Figure 1-14.

```
rect(50, 40, 75, 100);
```

By adding the `stroke()` and `fill()` functions *before* the shape is drawn, you can set the color. It's much like instructing your friend to use a specific pen to draw on the graph paper. You would have to tell your friend *before* he or she starting drawing, not after.

There is also the function `background()`, which sets a background color for the window where shapes will be rendered.

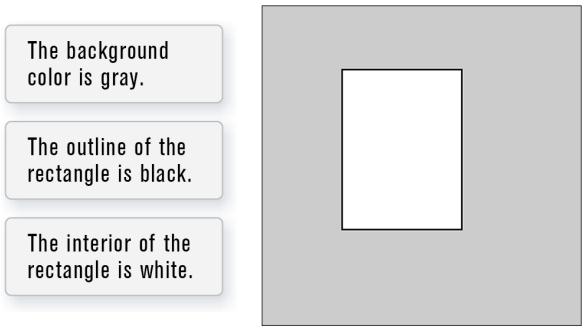


Figure 1-14

Example 1-1. Stroke and fill

```
background(255);
stroke(0);
fill(150);
rect(50, 50, 75, 100);
```

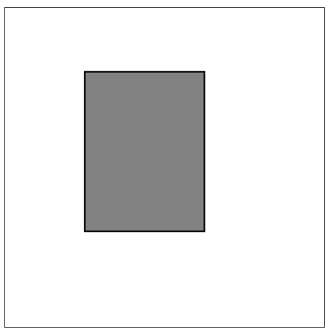


Figure 1-15

`stroke()` or `fill()` can be eliminated with the `noStroke()` or `noFill()` functions. Your instinct might be to say `stroke(0)` for no outline, however, it's important to remember that 0 is not "nothing," but rather denotes the color black. Also, remember not to eliminate both — with `noStroke()` and `noFill()`, nothing will appear!

Example 1-2. noFill()

```
background(255);
stroke(0);
noFill();
ellipse(60, 60, 100, 100);
```

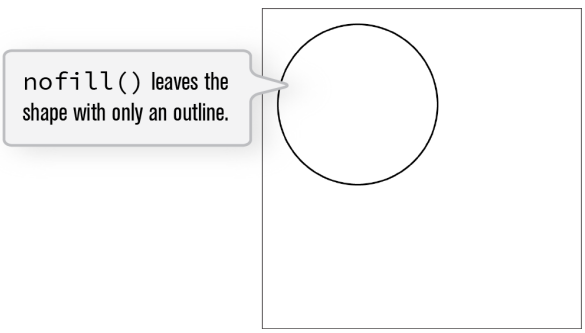


Figure 1-16

When you draw a shape, Processing will always use the most recently specified `stroke()` and `fill()`, reading the code from top to bottom. See Figure 1-17.

```
background(150);
stroke(0);
line(0, 0, 200, 200);
stroke(255);
noFill();
rect(25, 25, 75, 75);
```

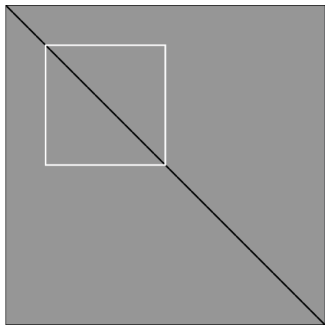


Figure 1-17

Exercise 1-4: Try to guess what the instructions would be for the following screenshot.

_____

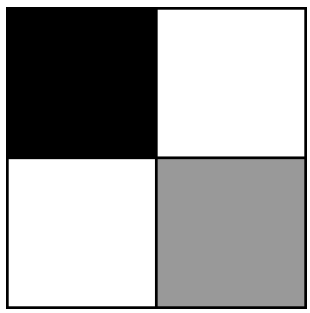
_____

_____

_____

_____

_____



1-4 RGB color

A nostalgic look back at graph paper helped you to learn the fundamentals for pixel locations and size. Now that it's time to study the basics of digital color, here's another childhood memory to get you started. Remember finger painting? By mixing three "primary" colors, any color could be generated. Swirling all colors together resulted in a muddy brown. The more paint you added, the darker it got.

Digital colors are also constructed by mixing three primary colors, but it works differently from paint. First, the primaries are different: red, green, and blue (i.e., "RGB" color). And with color on the screen, you're mixing light, not paint, so the mixing rules are different as well.

- Red + green = yellow
- Red + blue = purple
- Green + blue = cyan (blue-green)
- Red + green + blue = white
- No colors = black

This assumes that the colors are all as bright as possible, but of course, you have a range of color available, so some red plus some green plus some blue equals gray, and a bit of red plus a bit of blue equals dark purple.

While this may take some getting used to, the more you program and experiment with RGB color, the more it will become instinctive, much like swirling colors with your fingers. And of course you can't say "Mix some red with a bit of blue"; you have to provide an exact amount. As with grayscale, the individual color elements are expressed as ranges from 0 (none of that color) to 255 (as much as possible), and they are listed in the order red, green, and blue. You will get the hang of RGB color mixing through experimentation, but next I will cover some code using some common colors.

Note that the print version of this book will only show you black and white versions of each Processing sketch, but all sketches can be seen online in full color at *http://learningprocessing.com*. You can also see a color version of the tutorial on the Processing website (*https://processing.org/tutorials/color/*).

Example 1-3. RGB color

```
background(255);
noStroke();

fill(255, 0, 0);
ellipse(20, 20, 16, 16);

fill(127, 0, 0);
ellipse(40, 20, 16, 16);

fill(255, 200, 200);
ellipse(60, 20, 16, 16);
```

Bright red

Dark red

Pink (pale red).

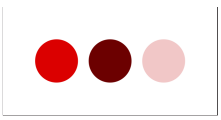


Figure 1-18

Processing also has a color selector to aid in choosing colors. Access this via "Tools" (from the menu bar) → "Color Selector." See Figure 1-19.

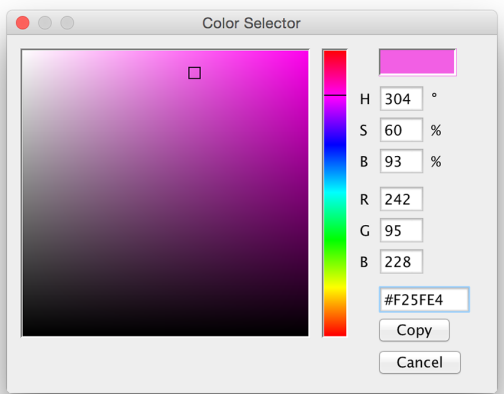


Figure 1-19

Exercise 1-5: Complete the following program. Guess what RGB values to use (you will be able to check your results in Processing after reading the next chapter). You could also use the color selector, shown in Figure 1-19.

```
fill(________,________,________);
ellipse(20, 40, 16, 16);

fill(________,________,________);
ellipse(40, 40, 16, 16);

fill(________,________,________);
ellipse(60, 40, 16, 16);
```

Bright blue

Dark purple

Yellow

Exercise 1-6: What color will each of the following lines of code generate?

```
fill(0, 100, 0);        ______________________________________

fill(100);              ______________________________________

stroke(0, 0, 200);      ______________________________________

stroke(225);            ______________________________________

stroke(255, 255, 0);    ______________________________________

stroke(0, 255, 255);    ______________________________________

stroke(200, 50, 50);    ______________________________________
```

1-5 Color transparency

In addition to the red, green, and blue components of each color, there is an additional optional fourth component, referred to as the color's "alpha." Alpha means opacity and is particularly useful when you want to draw elements that appear partially see-through on top of one another. The alpha values for an image are sometimes referred to collectively as the "alpha channel" of an image.

It's important to realize that pixels are not literally transparent; this is simply a convenient illusion that is accomplished by blending colors. Behind the scenes, Processing takes the color numbers and adds a percentage of one to a percentage of another, creating the optical perception of blending. (If you're interested in programming "rose-colored" glasses, this is where you would begin.)

Alpha values also range from 0 to 255, with 0 being completely transparent (i.e., zero percent opaque) and 255 completely opaque (i.e., 100 percent opaque). Example 1-4 shows a code example that is displayed in Figure 1-20.

Example 1-4. Opacity

```
background(0);
noStroke();

fill(0, 0, 255);
rect(0, 0, 100, 200);

fill(255, 0, 0, 255);
rect(0, 0, 200, 40);

fill(255, 0, 0, 191);
rect(0, 50, 200, 40);

fill(255, 0, 0, 127);
rect(0, 100, 200, 40);

fill(255, 0, 0, 63);
rect(0, 150, 200, 40);
```

No fourth argument means 100% opacity.

255 means 100% opacity.

75% opacity

50% opacity

25% opacity

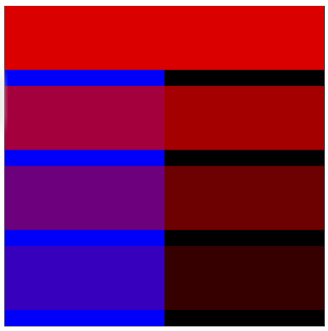


Figure 1-20

1-6 Custom color ranges

RGB color with ranges of 0 to 255 is not the only way you can handle color in Processing. Behind the scenes in the computer's memory, color is *always* talked about as a series of 24 bits (or 32 in the case of colors with an alpha). However, Processing will let you think about color any way you like, and translate any values into numbers the computer understands. For example, you might prefer to think of color as ranging from 0 to 100 (like a percentage). You can do this by specifying a custom `colorMode()`.

```
colorMode(RGB, 100);
```

With `colorMode()` you can set your own color range.

The above function says: "OK, I want to think about color in terms of red, green, and blue. The range of RGB values will be from 0 to 100."

Although it's rarely convenient to do so, you can also have different ranges for each color component:

```
colorMode(RGB, 100, 500, 10, 255);
```

Now I am saying "Red values range from 0 to 100, green from 0 to 500, blue from 0 to 10, and alpha from 0 to 255."

Finally, while you will likely only need RGB color for all of your programming needs, you can also specify colors in the HSB (hue, saturation, and brightness) mode. While HSB values also default to a range of 0 to 255, a common set of ranges (with brief explanation) are as follows:

- **Hue** — The shade of color itself (red, blue, orange, etc.) ranging from 0 to 360 (think of 360° on a color "wheel").
- **Saturation** — The vibrancy of the color, 0 to 100 (think of 50%, 75%, etc.).
- **Brightness** — The, well, brightness of the color, 0 to 100.

Exercise 1-7: Design a creature using simple shapes and colors. Draw the creature by hand using only points, lines, rectangles, and ellipses. Then attempt to write the code for the creature, using the Processing commands covered in this chapter: `point()`, `line()`, `rect()`, `ellipse()`, `stroke()`, *and* `fill()`. *In the next chapter, you will have a chance to test your results by running your code in Processing.*

Example 1-5 shows my version of Zoog, with the outputs shown in Figure 1-21.

Example 1-5. Zoog

```
background(255);
ellipseMode(CENTER);
rectMode(CENTER);
stroke(0);
fill(150);
rect(100,100,20,100);
fill(255);
ellipse(100,70,60,60);
fill(0);
ellipse(81,70,16,32);
ellipse(119,70,16,32);
stroke(0);
line(90,150,80,160);
line(110,150,120,160);
```

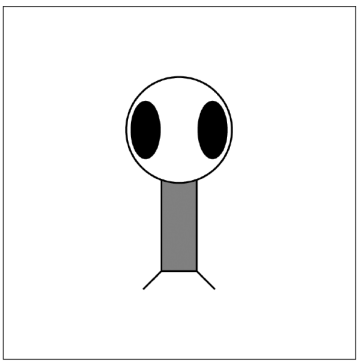


Figure 1-21

The sample answer is my Processing-born being, named Zoog. Over the course of the first nine chapters of this book, I will follow the course of Zoog's childhood. The fundamentals of programming will be demonstrated as Zoog grows up. You will first learn to display Zoog, then to make an interactive Zoog and animated Zoog, and finally to duplicate Zoog in a world of many Zoogs.

I suggest you design your own "thing" (note that there is no need to limit yourself to a humanoid or creature-like form; any programmatic pattern will do) and recreate all of the examples throughout the first nine chapters with your own design. Most likely, this will require you to change only a small portion (the shape rendering part) of each example. This process, however, should help solidify your understanding of the basic elements required for computer programs — *variables*, *conditionals*, *loops*, *functions*, *objects*, and *arrays* — and prepare you for when Zoog matures, leaves the nest, and ventures off into the more advanced topics from Chapter 10 onwards in this book.

This page intentionally left blank

2 Processing

Computers in the future may weigh no more than 1.5 tons.
—*Popular Mechanics*, 1949

Take me to your leader.
—Zoog, 2008

In this chapter:
- Downloading and installing Processing
- The Processing interface
- The Processing *sketchbook*
- Writing code
- Errors
- The Processing reference
- The Run button
- Your first sketch

2-1 Processing to the rescue

Now that you have conquered the world of primitive shapes and RGB color, you are ready to implement this knowledge in a real-world programming scenario. Happily, the environment you are going to use is Processing, free and open source software developed by Ben Fry and Casey Reas at the MIT Media Lab in 2001. (See this book's introduction for more about Processing's history.)

Processing's core library of functions for drawing graphics to the screen will provide immediate visual feedback and clues as to what the code is doing. And since its programming language employs all the same principles, structures, and concepts of other languages (specifically Java), everything you learn with Processing is *real* programming. It's not some pretend language to help you get started; it has all the fundamentals and core concepts that all languages have.

After reading this book and learning to program, you might continue to use Processing in your academic or professional life as a prototyping or production tool. You might also take the knowledge acquired here and apply it to learning other languages and authoring environments. You may, in fact, discover that programming is not your cup of tea; nonetheless, learning the basics will help you become more adept in collaborations with other designers and programmers.

It may seem like overkill to emphasize the *why* with respect to Processing. After all, the focus of this book is primarily on learning the fundamentals of computer programming in the context of computer graphics and design. It is, however, important to take some time to ponder the reasons behind selecting a programming language for a book, a class, a homework assignment, a web application, a software suite, and so forth. After all, now that you are going to start calling yourself a computer programmer at cocktail parties, this question will come up over and over again: I need programming in order to accomplish _____ project; which language and environment should I use?

For me, there is no correct answer to this question. Any language that you feel excited to try is a great language. And for a first try, Processing is particularly well suited. Its simplicity is ideal for a beginner. At the end of this chapter, you will be up and running with your first computational design and ready to learn the fundamental concepts of programming. But simplicity is not where Processing ends. A trip through the Processing online exhibition (*http://processing.org/exhibition*) will uncover a wide variety of beautiful and innovative projects developed entirely with Processing. By the end of this book, you will have all the tools and knowledge you need to take your ideas and turn them into real world software projects like those found in the exhibition. Processing is great both for learning and for producing; there are very few other environments and languages you can say that about.

2-2 How do I get Processing?

For the most part, this book will assume that you have a basic working knowledge of how to operate your personal computer. The good news, of course, is that Processing is available for free download. Head to processing.org and visit the download page. This book is designed to work with the Processing 3.0 series, I suggest downloading the latest version on the top of the page. If you're a Windows user, you will see two options: "Windows 32-bit" and "Windows 64-bit." The distinction is related to your machine's processor. If you're not sure which version of Windows you're running you'll find the answer by clicking the Start button, right-clicking Computer, and then clicking Properties. For Mac OS X, there is only one download option. There are also Linux versions available. Operating systems and programs change, of course, so if this paragraph is obsolete or out of date, the download page on the site includes information regarding what you need.

The Processing software will arrive as a compressed file. Choose a nice directory to store the application (usually "C:\Program Files\" on Windows and in "Applications" on Mac), extract the files there, locate the Processing executable, and run it.

Exercise 2-1: Download and install Processing.

2-3 The Processing application

The Processing development environment is a simplified environment for writing computer code, and it is just about as straightforward to use as simple text editing software (such as TextEdit or Notepad) combined with a media player. Each sketch (Processing programs are referred to as "sketches") has a name, a place where you can type code, and buttons for running sketches. See Figure 2-1. (At the time of this writing the version is Processing 3.0 alpha release 10 and so the version you download may look a little bit different.)

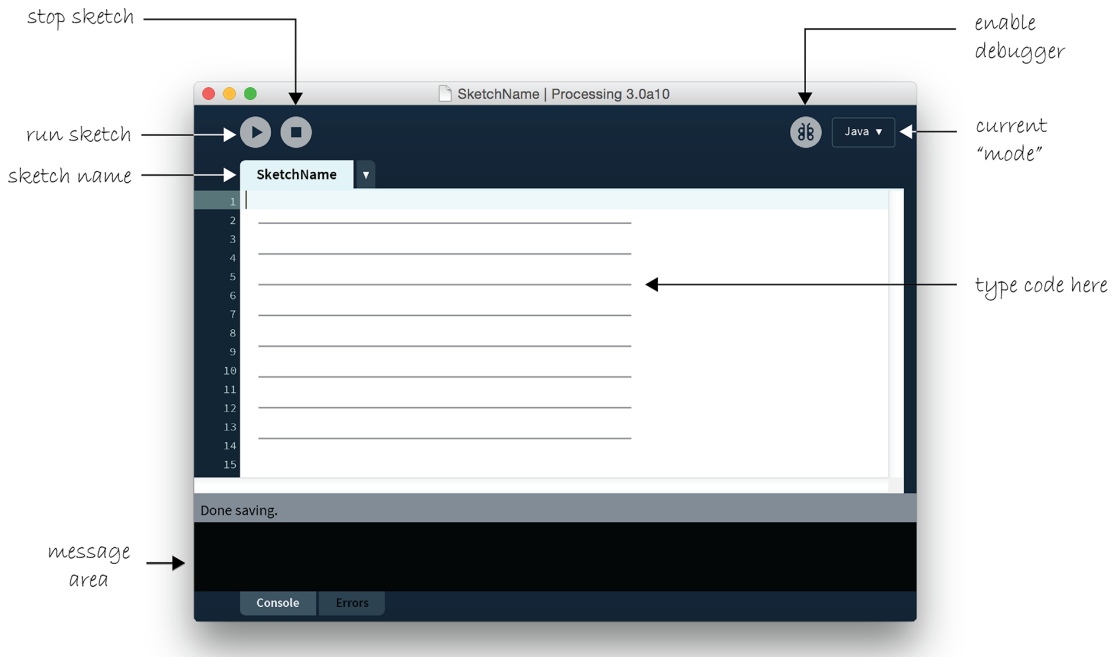


Figure 2-1

To make sure everything is working, it's a good idea to try running one of the Processing examples. Go to File → Examples → (pick an example, suggested: Topics → Drawing → ContinuousLines) as shown in Figure 2-2.

Once you have opened the example, click the Run button as indicated in Figure 2-1. If a new window pops open running the example, you're all set! If this does not occur, visit the troubleshooting FAQ (*https://github.com/processing/processing/wiki/troubleshooting*) and look for "Processing won't start!" for possible solutions.

Exercise 2-2: Open a sketch from the Processing examples and run it.

Processing programs can also be viewed full-screen (known as "Present mode" in Processing). This is available through the menu option: Sketch → Present (or by shift-clicking the Run button). Present will not make your sketch as big as the whole screen. If you want the sketch to cover your entire screen, you can use `fullScreen()` which I'll cover in more detail in the next section.

Under "Present," you'll also notice an option to "Tweak" your sketch, which will launch the program with a interface that allows you to tweak numbers on the fly. This can be useful for experimenting with the parameters of a sketch, from things as simple as the colors and dimensions of shapes to more complex elements of programs you'll learn about later in this book.

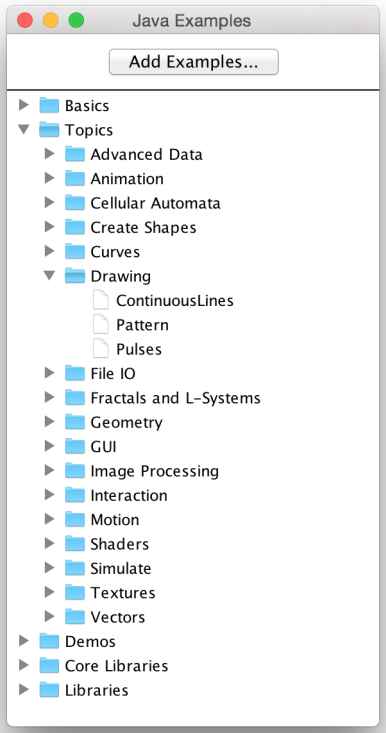


Figure 2-2

2-4 The sketchbook

Processing programs are informally referred to as *sketches*, in the spirit of quick graphics prototyping, and I will employ this term throughout the course of this book. The folder where you store your sketches is called your *sketchbook*. Technically speaking, when you run a sketch in Processing, it runs as an application on your computer. As you will see later in Chapter 21, Processing allows you to make platform-specific stand-alone applications from your sketches.

Once you have confirmed that the Processing examples work, you are ready to start creating your own sketches. Clicking the "new" button will generate a blank new sketch named by date. It's a good idea to "Save as" and create your own sketch name. (Note: Processing does not allow spaces or hyphens in sketch names, and your sketch name can't start with a digit.)

When you first ran Processing, a default "Processing" directory was created to store all sketches in the "My Documents" folder on Windows or in "Documents" on OS X. Although you can select any directory on your hard drive, this folder is the default. It's a pretty good folder to use, but it can be changed by opening the Processing preferences (which are available under the "File" menu).

Each Processing sketch consists of a folder (with the same name as your sketch) and a file with the extension "pde." If your Processing sketch is named *MyFirstProgram*, then you will have a folder named *MyFirstProgram* with a file *MyFirstProgram.pde* inside. This file is a plain text file that contains the source code. (Later you will see that Processing sketches can have multiple files with the "pde" extension, but for now, one will do.) Some sketches will also contain a folder called "data" where media elements used in the program, such as image files, sound clips, and so on, are stored.

Exercise 2-3: Type some instructions from Chapter 1 into a blank sketch. Note how certain words are colored. Run the sketch. Does it do what you thought it would?

2-5 Coding in Processing

It's finally time to start writing some code, using the elements discussed in Chapter 1. Let's go over some basic syntax rules. There are three kinds of statements you can write:

- Function calls
- Assignment operations
- Control structures

For now, every line of code will be a function call. See Figure 2-3. I will explore the other two categories in future chapters. Functions calls have a name, followed by a set of arguments enclosed in parentheses. Recalling Chapter 1, I used functions to describe how to draw shapes (I just called them "commands" or "instructions"). Thinking of a function call as a natural language sentence, the function name is the verb ("draw") and the arguments are the objects ("point 0,0") of the sentence. Each function call must always end with a semicolon. See Figure 2-4.

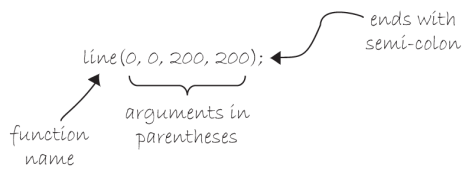


Figure 2-3

You have learned several functions already, including `background()`, `stroke()`, `fill()`, `noFill()`, `noStroke()`, `point()`, `line()`, `rect()`, `ellipse()`, `rectMode()`, and `ellipseMode()`. Processing will execute a sequence of functions one by one and finish by displaying the drawn result in a window. I forgot to mention one very important function in Chapter 1, however — `size()`. `size()` specifies the dimensions of the window you want to create and takes two arguments, width and height. If you want to show your sketch fullscreen, you can call `fullScreen()` instead of `size()`. Your sketch dimensions will match the resolution of your display. The `size()` or `fullScreen()` function should always be the first line of code in `setup()` and you can only have one of these in any given sketch.

```
void setup() {


  size(320, 240);
}
```

No code can go here before `size()`!

Opens a window of width 320 and height 240.

Here is `fullScreen()`.

```
void setup() {

  fullScreen();
}
```

No code can go here before `fullScreen()`!

Opens a fullscreen window.

Let's write a first example (see Figure 2-4).

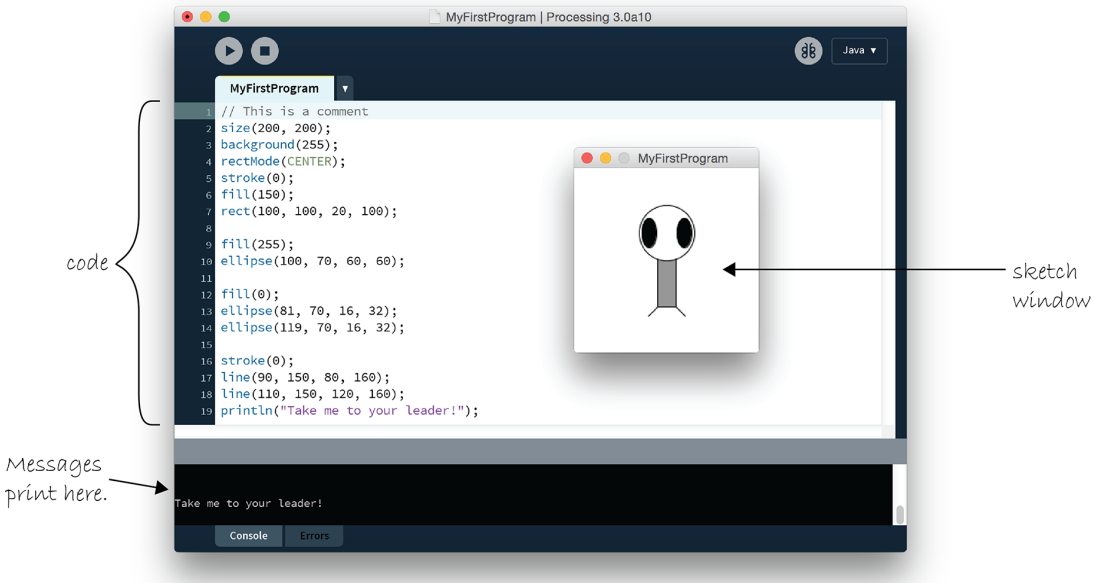


Figure 2-4

There are a few additional items to note.

- The Processing text editor will color known words (sometimes referred to as *reserved* words or *keywords*). These words, for example, are the drawing functions available in the Processing library, built-in variables (I will look closely at the concept of *variables* in Chapter 3) and constants, as well as certain words that are inherited from the Java programming language.
- Sometimes it's useful to display text information in the Processing message window (located at the bottom). This is accomplished using the `println()` function. The `println()` function takes one or more arguments, whatever you want to print to the message window. In this case (as shown in Figure 2-4) I'm printing the string of characters enclosed in quotes: "Take me to your leader!" (more about text in Chapter 17). This ability to print to the message window comes in handy when attempting to *debug* the values of variables. There's also a special button for debugging, the little insect in the top right, and I'll reference this again in Chapter 11.
- The number in the bottom left corner indicates what line number in the code is selected. You can also see the line numbers to the left of your code.

- You can write *comments* in your code. Comments are lines of text that Processing ignores when the program runs. You should use them as reminders of what the code means, a bug you intend to fix, a to-do list of items to be inserted, and so on. Comments on a single line are created with two forward slashes, `//`. Comments over multiple lines are marked by `/*` followed by the comments and ending with `*/`.
- Processing starts, by default, in what's known as "Java" mode. This is the core use of Processing where your code is written in the Java programming language. There are other modes, notably Python Mode, which allow you create Processing sketches in the Python programming language. You can explore these modes by clicking the mode button as indicated in Figure 2-4.

```
// This is a comment on one line.

/* This is a comment
that spans several
lines of code. */
```

A quick word about comments. You should get in the habit right now of writing comments in your code. Even though your sketches will be very simple and short at first, you should put comments in for everything. Code is very hard to read and understand without comments. You do not need to have a comment for every line of code, but the more you include, the easier a time you will have revising and reusing your code later. Comments also force you to understand how code works as you're programming. If you do not know what you're doing, how can you write a comment about it?

Comments will not always be included in the text here. This is because I find that, unlike in an actual program, code comments are hard to read in a book. Instead, this book will often use code "hints" for additional insight and explanations. If you look at the book's examples on the website, though, comments will always be included. So, I can't emphasize it enough: write comments!

```
// Draw a diagonal line starting at upper left
line(0, 0, 100, 100);
```

A helpful comment about this code!

Exercise 2-4: Create a blank sketch. Take your code from the end of Chapter 1 and type it in the Processing window. Add comments to describe what the code is doing. Add a `println()` *statement to display text in the message window. Save the sketch. Press the Run button. Does it work or do you get an error?*

2-6 Errors

The previous example only works because I did not make any errors or typos. Over the course of a programmer's life, this is quite a rare occurrence. Most of the time, your first push of the Run button will not be met with success. Let's examine what happens when you make a mistake in the code in Figure 2-5.

Figure 2-5 shows what happens when you have a typo — "elipse" instead of "ellipse" on line 9. Errors are noted in the code itself with a red squiggly line underneath where Processing believes the mistake to be. This particular message is fairly friendly, telling you that Processing has never heard of the function

"elipse." This can easily be corrected by fixing the spelling. If there is an error in the code when the Run button is pressed, Processing will not open the sketch window, and will instead highlight the error message. Not all Processing error messages are so easy to understand, and I will continue to cover other errors throughout the course of this book. An appendix on common errors in Processing is also included at the end of the book.

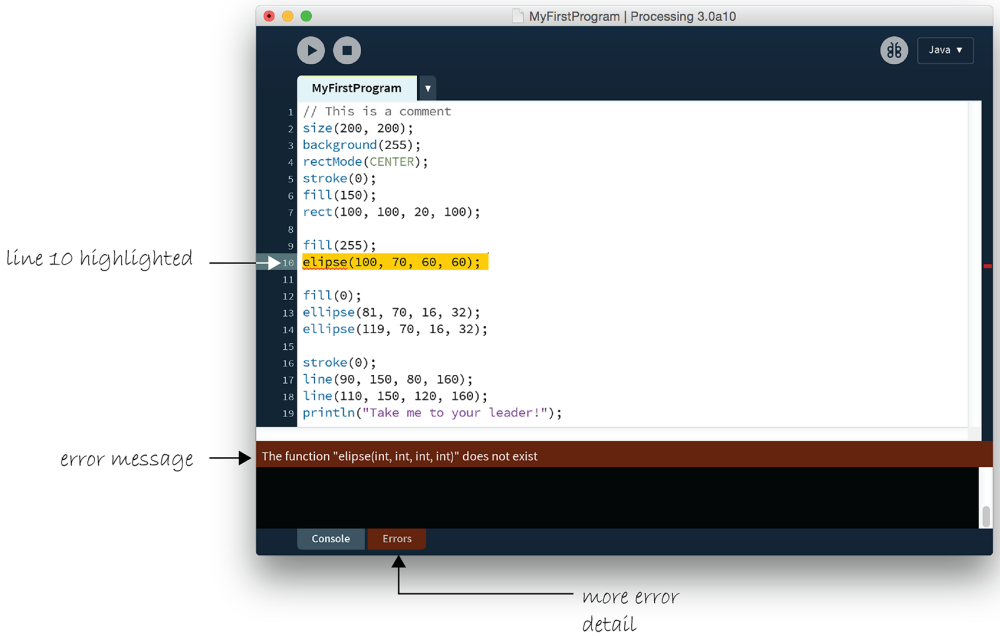


Figure 2-5

Processing is case sensitive!

Lower versus upper case matters. If you type `Ellipse` instead of `ellipse`, that will also be considered an error.

In this instance, there was only one error. If multiple errors occur, Processing will only alert you to the first one it finds when you press run. However, a complete list of errors can always be found in the errors console at the bottom as noted in Figure 2-5. Dealing with just one error at a time is much less stressful, however, so this further emphasizes the importance of incremental development discussed in the book's introduction. If you only implement one feature at a time, you can only make one mistake at a time.

Exercise 2-5: Try to make some errors happen on purpose. Are the error messages what you expect?

Exercise 2-6: Fix the errors in the following code.

```
size(200, 200;          ________________________________________

background();            ________________________________________

stroke 255;             ________________________________________

fill(150)               ________________________________________

rectMode(center);       ________________________________________

rect(100, 100, 50);     ________________________________________
```

2-7 The Processing reference

The functions I have demonstrated — `ellipse()`, `line()`, `stroke()`, and so on — are all part of *Processing*'s library. How do you know that "ellipse" isn't spelled "elipse," or that `rect()` takes four arguments (an x-coordinate, a y-coordinate, a width, and a height)? A lot of these details are intuitive, and this speaks to the strength of Processing as a beginner's programming language. Nevertheless, the only way to know for sure is by reading the online reference. While I will cover many of the elements from the reference throughout this book, it is by no means a substitute for the reference, and both will be required for you to learn Processing.

The reference for Processing can be found online at the official website (*processing.org*) under the "reference" link. There, you can browse all of the available functions by category or alphabetically. If you were to visit the page for `ellipse()`, for example, you would find the explanation shown in Figure 2-6.

As you can see, the reference page offers full documentation for the function `rect()`, including:

- **Name** — The name of the function.
- **Examples** — Example code (and visual result, if applicable).
- **Description** — A friendly description of what the function does.
- **Syntax** — Exact syntax of how to write the function.
- **Parameters** — These are the elements that go inside the parentheses. It tells you what kind of data you put in (a number, character, etc.) and what that element stands for. (This will become clearer as I explain more in future chapters.) These are also sometimes referred to as *arguments*.

- **Returns** — Sometimes a function sends something back to you when you call it (e.g., instead of asking a function to perform a task such as draw a circle, you could ask a function to add two numbers and *return* the answer to you). Again, this will become more clear later.
- **Related** — A list of functions often called in connection with the current function.

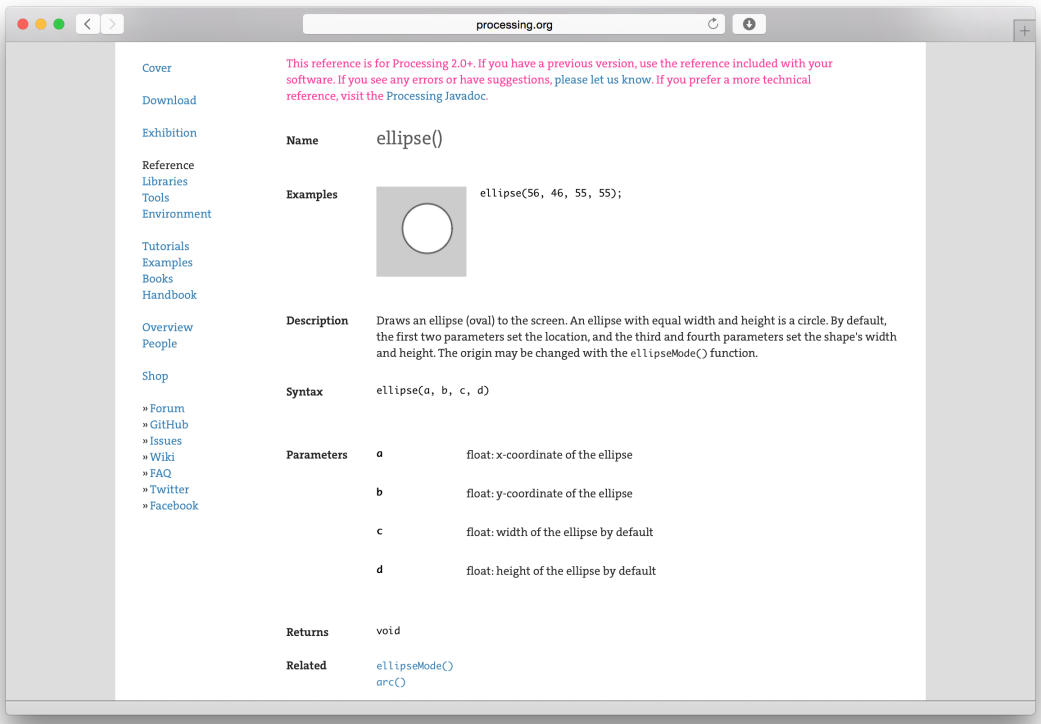


Figure 2-6

Processing also has a very handy "find in reference" option. Double-click on any keyword to select it and go to Help → Find in Reference (or select the keyword and hit Shift+Command+F (Mac) or Ctrl+Shift +F (PC)).

Exercise 2-7: Using the Processing reference, try writing a program that uses two functions I have not yet covered in this book. Stay within the "Shape" and "Color (setting)" categories.

Exercise 2-8: Using the reference, find a function that allows you to alter the thickness of a line. What arguments does the function take? Write example code that draws a line one pixel wide, then five pixels wide, then 10 pixels wide.

2-8 The Run button

One of the nice qualities of Processing is that all one has to do to run a program is press the Run button. The design is similar to a media "play" button you might find when *playing* animations, movies, music, and other forms of media. Processing programs output media in the form of real-time computer graphics, so why not just *play* them too?

Nevertheless, it's important to take a moment and consider the fact that what I am doing here is not the same as what happens when an audio or video plays. Processing programs start out as text, they are translated into machine code, and then executed to run. All of these steps happen in sequence when the Run button is pressed. Let's examine these steps one by one, relaxed in the knowledge that Processing handles the hard work for you.

1. **Translate to Java.** Processing is really Java (this will become more evident in a detailed discussion in Chapter 23). In order for your code to run on your machine, it must first be translated to Java code.
2. **Compile into Java byte code.** The Java code created in Step 1 is just another text file (with the .java extension instead of .pde). In order for the computer to understand it, it needs to be translated into machine language. This translation process is known as compilation. If you were programming in a different language, such as C, the code would compile directly into machine language specific to your operating system. In the case of Java, the code is compiled into a special machine language known as Java byte code. It can run on different platforms as long as the machine is running a "Java Virtual Machine." Although this extra layer can sometimes cause programs to run a bit slower than they might otherwise, being cross-platform is a great feature of Java. For more on how this works, visit the official Java website (*http://www.oracle.com/technetwork/java/index.html*) or consider picking up a book on Java programming (after you have finished with this one).
3. **Execution.** The compiled program ends up in a JAR file. A JAR is a Java archive file that contains compiled Java programs ("classes"), images, fonts, and other data files. The JAR file is executed by the Java Virtual Machine and is what causes the display window to appear.

2-9 Your first sketch

Now that you have downloaded and installed Processing, understand the basic menu and interface elements, and are familiar with the online reference, you are ready to start coding. As I briefly mentioned in Chapter 1, the first half of this book will follow one example that illustrates the foundational elements of programming: *variables*, *conditionals*, *loops*, *functions*, *objects*, and *arrays*. Other examples will be included along the way, but following just one will reveal how the basic elements behind computer programming build on each other.

The example will follow the story of our new friend Zoog, beginning with a static rendering with simple shapes. Zoog's development will include mouse interaction, motion, and cloning into a population of many Zoogs. While you are by no means required to complete every exercise of this book with your own alien form, it can be helpful to start with an idea and after each chapter, expand the functionality of your sketch with the programming concepts that are explored. If you're at a loss for an idea, then just draw your own little alien, name it Gooz, and get programming! See Figure 2-7.

Example 2-1. Zoog again

```
size(200, 200); // Set the size of the window
background(255); // Draw a white background

// Set ellipses and rects to CENTER mode
ellipseMode(CENTER);
rectMode(CENTER);

// Draw Zoog's body
stroke(0);
fill(150);
rect(100, 100, 20, 100);

// Draw Zoog's head
fill(255);
ellipse(100, 70, 60, 60);

// Draw Zoog's eyes
fill(0);
ellipse(81, 70, 16, 32);
ellipse(119, 70, 16, 32);

// Draw Zoog's legs
stroke(0);
line(90, 150, 80, 160);
line(110, 150, 120, 160);
```

Zoog's body

Zoog's head

Zoog's eyes

Zoog's legs

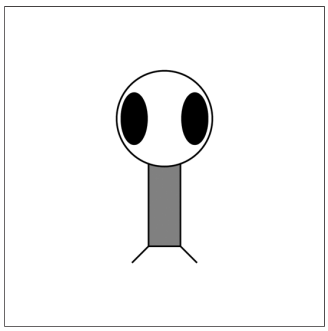


Figure 2-7

Let's pretend, just for a moment, that you find this Zoog design to be so astonishingly gorgeous that you just can't wait to see it displayed on your computer screen. (Yes, I am aware this may require a fairly significant suspension of disbelief.) To run any and all code examples found in this book, you have two choices:

- Retype the code manually.
- Visit the book's website (*http://learningprocessing.com*), find the example by its number, and copy/paste (or download) the code.

Certainly option #2 is the easier and less time-consuming one, and I recommend you use the site as a resource for seeing sketches running in real time and for grabbing code examples. Nonetheless, as you start learning, there is real value in typing the code yourself. Your brain will sponge up the syntax and logic as you type, and you will learn a great deal by making mistakes along the way. Not to mention simply running the sketch after entering each new line of code will eliminate any mystery as to how the sketch works.

You will know best when you are ready for copy/paste. Keep track of your progress, and if you start running a lot of examples without feeling comfortable with how they work, try going back to manual typing.

Exercise 2-9: Using what you designed in Chapter 1, implement your own screen drawing, using only 2D primitive shapes — `arc()`, `curve()`, `ellipse()`, `line()`, `point()`, `quad()`, `rect()`, `triangle()` *— and basic color functions —* `background()`, `colorMode()`, `fill()`, `noFill()`, `noStroke()`, *and* `stroke()`. *Remember to use* `size()` *to specify the dimensions of your window or* `fullScreen()` *to have your sketch cover your entire display. Suggestion: Play the sketch after typing each new line of code. Correct any errors or typos along the way.*

This page intentionally left blank

3 Interaction

Always remember that this whole thing was started with a dream and a mouse.
—Walt Disney

The quality of the imagination is to flow and not to freeze.
—Ralph Waldo Emerson

In this chapter:
- The *flow* of a computer program
- The meaning behind `setup()` and `draw()`
- Mouse interaction
- Your first *dynamic* Processing sketch
- Handling events, such as mouse clicks and key presses

3-1 Go with the flow

If you have ever played a computer game, interacted with a digital art installation, or watched a screensaver at three in the morning, you have probably given very little thought to the fact that the software that runs these experiences happens over a *period of time*. The game starts, you find the secret treasure hidden in magical rainbow land, defeat the the scary monster who-zee-ma-whats-it, achieve a high score, and the game ends.

What I want to focus on in this chapter is that *flow* over time. A game begins with a set of initial conditions: you name your character, you start with a score of zero, and you start on level one. Let's think of this part as the program's *SETUP*. After these conditions are initialized, you begin to play the game. At every instant, the computer checks what you are doing with the mouse, calculates all the appropriate behaviors for the game characters, and updates the screen to render all the game graphics. This cycle of calculating and drawing happens over and over again, ideally 30 or more times per second for a smooth animation. Let's think of this part as the program's *DRAW*.

This concept is crucial to your ability to move beyond static designs (as in Chapter 2) with Processing.

1. Set starting conditions for the program one time.
2. Do something over and over and over and over (and over...) again until the program quits.

Consider how you might go about running a race.

1. Put on your sneakers and stretch. Just do this once, OK?
2. Put your right foot forward, then your left foot. Repeat this over and over as fast as you can.
3. After 26 miles, quit.

Exercise 3-1: In English, write out the flow for a simple computer game, such as Pong. If you're not familiar with Pong, visit: http://en.wikipedia.org/wiki/Pong.

_____

_____

_____

_____

3-2 Our good friends, `setup()` and `draw()`

Now that you are good and exhausted from running marathons in order to better learn programming, you can take this newfound knowledge and apply it to your first *dynamic* Processing sketch. Unlike Chapter 2's static examples, this program will draw to the screen continuously (i.e., until the user quits). This is accomplished by writing two "blocks of code": `setup()` and `draw()`. Technically speaking, `setup()` and `draw()` are functions. I will get into a longer discussion of writing your own functions in a later chapter; for now, you can understand them to be two sections where you write code.

What is a block of code?

A block of code is any code enclosed within curly brackets.

```
{
  A block of code
}
```

Blocks of code can be nested within each other, too.

```
{
  A block of code
  {
    A block inside a block of code
  }
}
```

This is an important construct as it allows you to separate and manage code as individual pieces of a larger puzzle. A programming convention is to indent the lines of code within each block to make the code more readable. Processing will do this for you via the menu option Edit → Auto-Format. Getting comfortable with organizing your code into blocks while more complex logic will prove crucial in future chapters. For now, you only need to look at two simple blocks: `setup()` and `draw()`.

Let's look at what will surely be strange-looking syntax for `setup()` and `draw()`. See Figure 3-1.

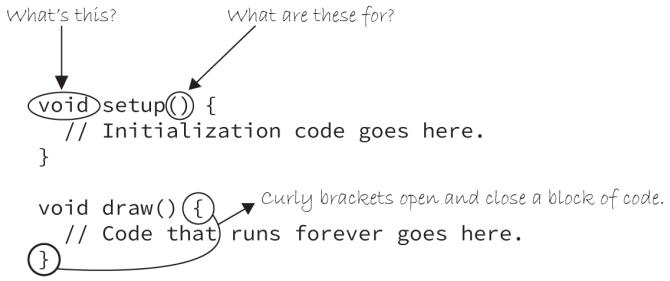


Figure 3-1

Admittedly, there is a lot of stuff in Figure 3-1 that you are not entirely ready to learn about. I have covered that the curly brackets indicate the beginning and end of a block of code, but why are there parentheses after "setup" and "draw"? Oh, and, my goodness, what is this "void" all about? It makes me feel sad inside! For now, you have to decide to feel comfortable with not knowing everything all at once, and that these important pieces of syntax will start to make sense in future chapters as more concepts are revealed.

For now, the key is to focus on how Figure 3-1's structures control the flow of a program. This is shown in Figure 3-2.

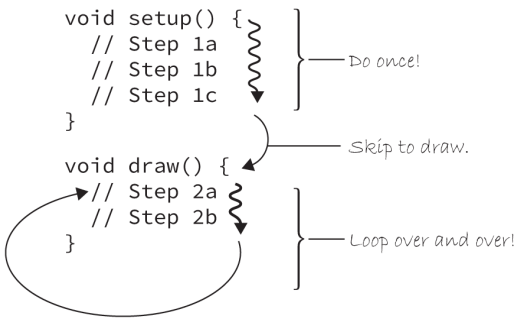


Figure 3-2

How does it work? When you run the program, it will follow the instructions precisely, executing the steps in `setup()` first, and then move on to the steps in `draw()`. The order ends up being something like:

1a, 1b, 1c, 2a, 2b, 2a, 2b, 2a, 2b, 2a, 2b, 2a, 2b, 2a, 2b...

Now, I can rewrite the Zoog example as a dynamic sketch. See Example 3-1.

Example 3-1. Zoog as dynamic sketch

```
void setup() {
  // Set the size of the window
  size(200, 200);
}

void draw() {
  // Draw a white background
  background(255);

  // Set CENTER mode
  ellipseMode(CENTER);
  rectMode(CENTER);

  // Draw Zoog's body
  stroke(0);
  fill(150);
  rect(100, 100, 20, 100);

  // Draw Zoog's head
  stroke(0);
  fill(255);
  ellipse(100, 70, 60, 60);

  // Draw Zoog's eyes
  fill(0);
  ellipse(81, 70, 16, 32);
  ellipse(119, 70, 16, 32);

  // Draw Zoog's legs
  stroke(0);
  line(90, 150, 80, 160);
  line(110, 150, 120, 160);
}
```

`setup()` runs first, once and only once. `size()` should always be first line of `setup()` since Processing will not be able to do anything before the window size is specified.

`draw()` loops continuously until you close the sketch window.

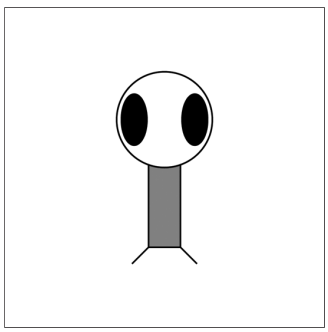


Figure 3-3

Take the code from Example 3-1 and run it in Processing. Strange, right? You will notice that nothing in the window changes. This looks identical to a *static* sketch! What is going on? All this discussion for nothing?

Well, if you examine the code, you will notice that nothing in the `draw()` function *varies*. Each time through the loop, the program cycles through the code and executes the identical instructions. So, yes, the program is running over time redrawing the window, but it looks static since it draws the same thing each time!

Exercise 3-2: Redo the drawing you created at the end of Chapter 2 as a dynamic program. Even though it will look the same, feel good about your accomplishment!

3-3 Variation with the mouse

Consider this: What if, instead of typing a number into one of the drawing functions, you could type "the mouse's x location" or "the mouse's y location."

```
line(the mouse's x location, the mouse's y location, 100, 100);
```

In fact, you can, only instead of the more descriptive language, you must use the keywords `mouseX` and `mouseY`, indicating the horizontal or vertical position of the mouse cursor.

Example 3-2. mouseX *and* mouseY

```
void setup() {
  size(200, 200);
}


void draw() {
  background(255);


  // Body
  stroke(0);
  fill(175);
  rectMode(CENTER);
  rect(mouseX, mouseY, 50, 50);
}
```

Try moving `background()` to `setup()` and see the difference! (Exercise 3-3)

`mouseX` is a keyword that the sketch replaces with the horizontal position of the mouse. `mouseY` is a keyword that the sketch replaces with the vertical position of the mouse.

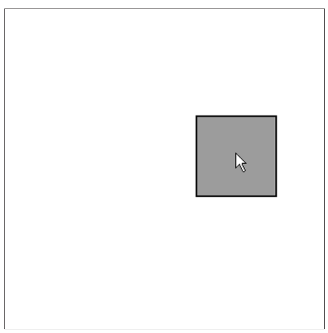
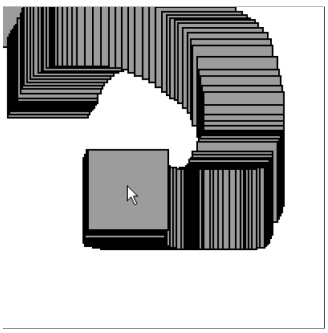


Figure 3-4

Exercise 3-3: Explain why you see a trail of rectangles if you move `background()` *to* `setup()`*, leaving it out of* `draw()`.



An invisible line of code

If you are following the logic of `setup()` and `draw()` closely, you might arrive at an interesting question: *When does* Processing *actually display the shapes in the window? When do the new pixels appear?*

On first glance, one might assume the display is updated for every line of code that includes a drawing function. If this were the case, however, you would see the shapes appear onscreen one at a time. This would happen so fast that you would hardly notice each shape appearing individually. However, when the window is erased every time `background()` is called, a somewhat unfortunate and unpleasant result would occur: flicker.

Processing solves this problem by updating the window only at the end of every cycle through `draw()`. It's as if there were an invisible line of code that renders the window at the end of the `draw()` function.

```
void draw() {
  // All of your code
  // Update Display Window – invisible line of code you don't see
}
```

This process is known as *double-buffering* and, in a lower-level environment, you may find that you have to implement it yourself. Again, I'd like to take a moment to thank Processing for making our introduction to programming friendlier and simpler by taking care of this for you and me.

It should also be noted that any colors you have set with `stroke()` or `fill()` carry over from one cycle through `draw()` to the next.

I could push this idea a bit further and create an example where a more complex pattern (multiple shapes and colors) is controlled by `mouseX` and `mouseY` position. For example, I can rewrite Zoog to follow the mouse. Note that the center of Zoog's body is located at the exact location of the mouse (`mouseX`, `mouseY`), however, other parts of Zoog's body are drawn relative to the mouse. Zoog's head, for example, is located at (`mouseX`, `mouseY-30`). The following example only moves Zoog's body and head, as shown in Figure 3-5.

Example 3-3. Zoog as dynamic sketch with variation

```
void setup() {
  size(200, 200); // Set the size of the window
}

void draw() {
  background(255); // Draw a white background

  // Set ellipses and rects to CENTER mode
  ellipseMode(CENTER);
  rectMode(CENTER);

  // Draw Zoog's body
  stroke(0);
  fill(175);
  rect(mouseX, mouseY, 20, 100);

  // Draw Zoog's head
  stroke(0);
  fill(255);
  ellipse(mouseX, mouseY-30, 60, 60);

  // Draw Zoog's eyes
  fill(0);
  ellipse(81, 70, 16, 32);
  ellipse(119, 70, 16, 32);

  // Draw Zoog's legs
  stroke(0);
  line(90, 150, 80, 160);
  line(110, 150, 120, 160);
}
```

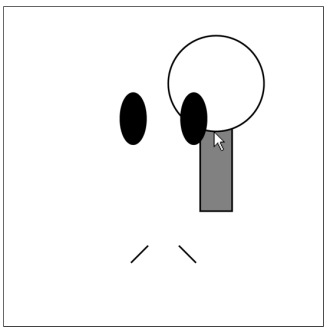


Figure 3-5

Zoog's body is drawn at the location (`mouseX`, `mouseY`).

Zoog's head is drawn above the body at the location (`mouseX`, `mouseY-30`).

Exercise 3-4: Complete Zoog so that the rest of its body moves with the mouse.

```
// Draw Zoog's eyes
fill(0);

ellipse(_______,_______, 16, 32);

ellipse(_______,_______, 16, 32);

// Draw Zoog's legs
stroke(0);

line(_______,_______,_______,_______);

line(_______,_______,_______,_______);
```

Exercise 3-5: Recode your design so that shapes respond to the mouse (by varying color and location).

In addition to `mouseX` and `mouseY`, you can also use `pmouseX` and `pmouseY`. These two keywords stand for the *previous* `mouseX` and `mouseY` locations, that is, where the mouse was the last time the sketch cycled through `draw()`. This allows for some interesting interaction possibilities. For example, let's consider what happens if you draw a line from the previous mouse location to the current mouse location, as illustrated in the diagram in Figure 3-6.

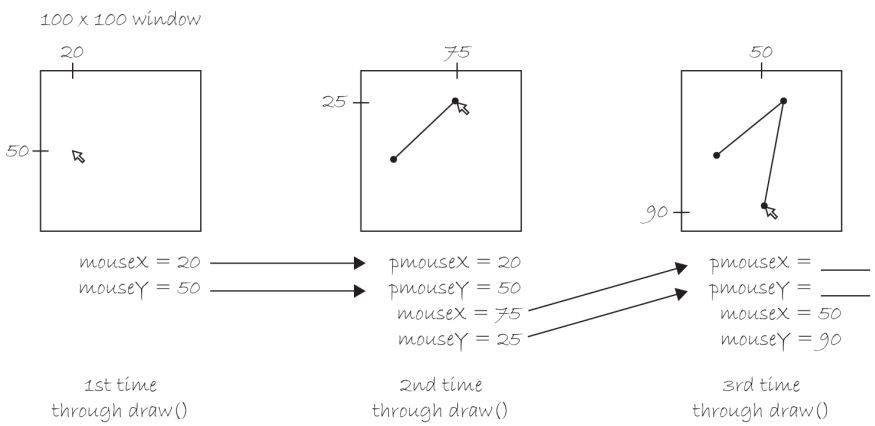


Figure 3-6

Exercise 3-6: Fill in the blank in Figure 3-6.

By connecting the previous mouse location to the current mouse location with a line each time through `draw()`, I am able to render a continuous line that follows the mouse. See Figure 3-7.

Example 3-4. Drawing a continuous line

```
void setup() {
  size(200, 200);
  background(255);
}

void draw() {
  stroke(0);
  line(pmouseX, pmouseY, mouseX, mouseY);
}
```

Draw a line from previous mouse location to current mouse location.

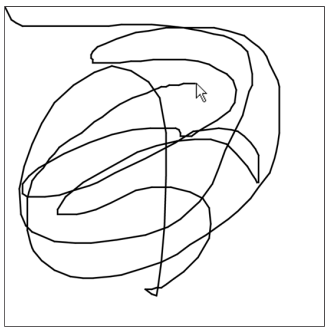


Figure 3-7

Exercise 3-7: Update Exercise 3-4 on page 40 so that the faster the user moves the mouse, the wider the drawn line. Hint: look up `strokeWeight()` *in the Processing reference (https://processing.org/reference/strokeWeight_.html).*

The formula for calculating the speed of the mouse's horizontal motion is the absolute value of the difference between `mouseX` and `pmouseX`. The absolute value of a number is defined as that number without its sign:

- The absolute value of -2 is 2.
- The absolute value of 2 is 2.

In Processing, you can get the absolute value of the number by placing it inside the `abs()` function, that is `abs(-5)` equals 5. The speed at which the mouse is moving is therefore:

```
float mouseSpeed = abs(mouseX - pmouseX);
```

Fill in the blank below and then try it out in Processing!

```
stroke(0);

_____________________________(_____________);
line(pmouseX, pmouseY, mouseX, mouseY);
```

3-4 Mouse clicks and key presses

You are well on your way to creating dynamic, interactive Processing sketches through the use the `setup()` and `draw()` framework and the `mouseX` and `mouseY` keywords. A crucial form of interaction, however, is missing — clicking the mouse!

In order to learn how to have something happen when the mouse is clicked, you need to return to the flow of the program. You know `setup()` happens once and `draw()` loops forever. When does a mouse click occur? Mouse presses (and key presses) are considered *events* in Processing. If you want something to happen (such as "the background color changes to red") when the mouse is clicked, you need to add a third block of code to handle this event.

This event "function" will tell the program what code to execute when an event occurs. As with `setup()`, the code will occur once and only once. That is, once and only once for each occurrence of the event. An event, such as a mouse click, can happen multiple times of course!

These are the two new functions you need:

- `mousePressed()` — Handles mouse clicks.
- `keyPressed()` — Handles key presses.

The following example uses both event functions, adding squares whenever the mouse is pressed and clearing the background whenever a key is pressed.

Example 3-5. `mousePressed()` **and** `keyPressed()`

```
void setup() {
  size(200, 200);
  background(255);
}

void draw() {

}

void mousePressed() {
  stroke(0);
  fill(175);
  rectMode(CENTER);
  rect(mouseX, mouseY, 16, 16);
}

void keyPressed() {
  background(255);
}
```

Nothing happens in `draw()` in this example!

Whenever a user clicks the mouse the code written inside `mousePressed()` is executed.

Whenever a user presses a key the code written inside `keyPressed()` is executed.

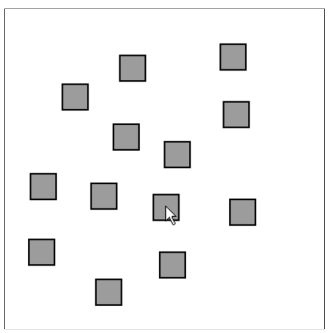


Figure 3-8

In Exercise 3-5 on page 40, I have four functions that describe the program's flow. The program starts in `setup()` where the size and background are initialized. It continues into `draw()`, looping endlessly. Since `draw()` contains no code, the window will remain blank. However, I have added two new functions: `mousePressed()` and `keyPressed()`. The code inside these functions sits and waits. When the user clicks the mouse (or presses a key), it springs into action, executing the enclosed block of instructions once and only once.

Exercise 3-8: Add `background(255);` *to the* `draw()` *function. Why does the program stop working?*

I am now ready to bring all of these elements together for Zoog.

- Zoog's entire body will follow the mouse.
- Zoog's eye color will be determined by mouse location.
- Zoog's legs will be drawn from the previous mouse location to the current mouse location.
- When the mouse is clicked, a message will be displayed in the message window: "Take me to your leader!"

Note the addition in Exercise 3-6 on page 41 of the function `frameRate()`. `frameRate()`, which requires a value of at least one, enforces the speed at which Processing will cycle through `draw()`. `frameRate(30)`, for example, means 30 frames per second, a traditional speed for computer animation. If you do not include `frameRate()`, Processing will attempt to run the sketch at 60 frames per second. Since computers run at different speeds, `frameRate()` is used to make sure that your sketch is consistent across multiple computers.

This frame rate is just a maximum, however. If your sketch has to draw one million rectangles, it may take a long time to finish the draw cycle and run at a slower speed.

Example 3-6. Interactive Zoog

```
void setup() {
  // Set the size of the window
  size(200, 200);
  frameRate(30);
}

void draw() {
  // Draw a white background
  background(255);

  // Set ellipses and rects to CENTER mode
  ellipseMode(CENTER);
  rectMode(CENTER);

  // Draw Zoog's body
  stroke(0);
  fill(175);
  rect(mouseX, mouseY, 20, 100);

  // Draw Zoog's head
  stroke(0);
  fill(255);
  ellipse(mouseX, mouseY-30, 60, 60);

  // Draw Zoog's eyes
  fill(mouseX, 0, mouseY);
  ellipse(mouseX-19, mouseY-30, 16, 32);
  ellipse(mouseX+19, mouseY-30, 16, 32);

  // Draw Zoog's legs
  stroke(0);
  line(mouseX-10, mouseY+50, pmouseX-10, pmouseY + 60);
  line(mouseX+10, mouseY+50, pmouseX+10, pmouseY + 60);
}

void mousePressed() {
  println("Take me to your leader!");
}
```

The frame rate is set to 30 frames per second.

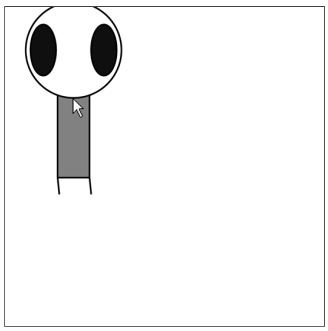


Figure 3-9

The eye color is determined by the mouse location.

The legs are drawn according to the mouse location and the previous mouse location.

Lesson One Project

(You may have completed much of this project already via the exercises in Chapter 1–Chapter 3. This project brings all of the elements together. You could either start from scratch with a new design or use elements from the exercises.)

1. Design a static screen drawing using RGB color and primitive shapes.
2. Make the static screen drawing dynamic by having it interact with the mouse. This might include shapes following the mouse, changing their size according to the mouse, changing their color according to the mouse, and so on.

Use the space provided below to sketch designs, notes, and pseudocode for your project.

This page intentionally left blank