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 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:

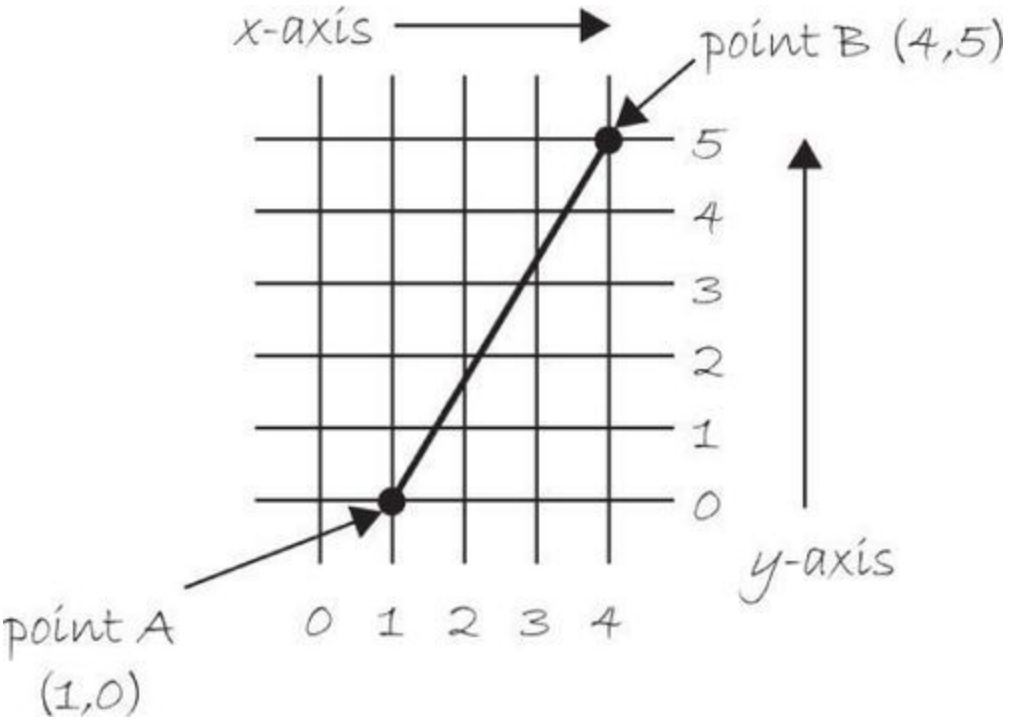


FIGURE 1-1

```
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.

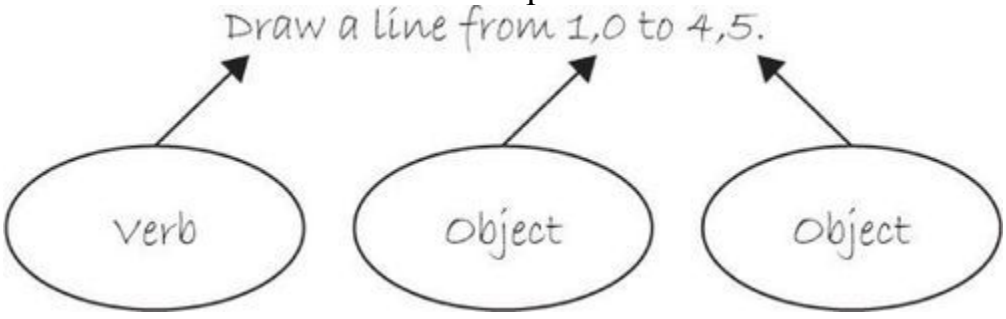


FIGURE 1-2

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.

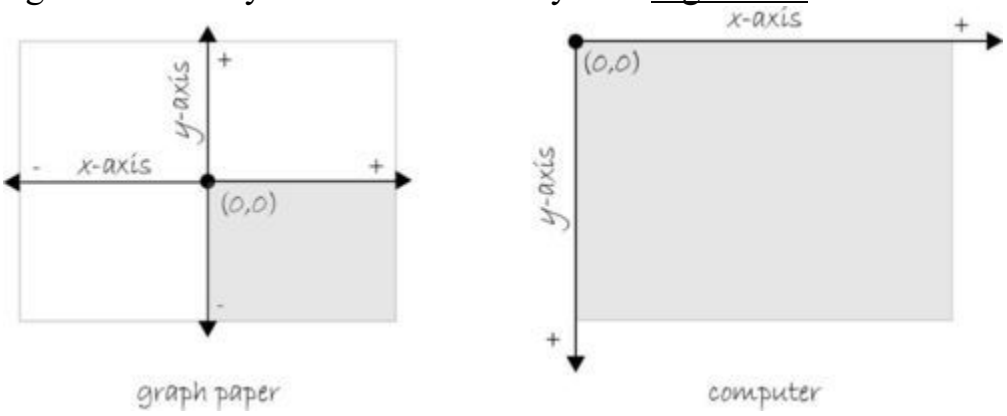


FIGURE 1-3

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.

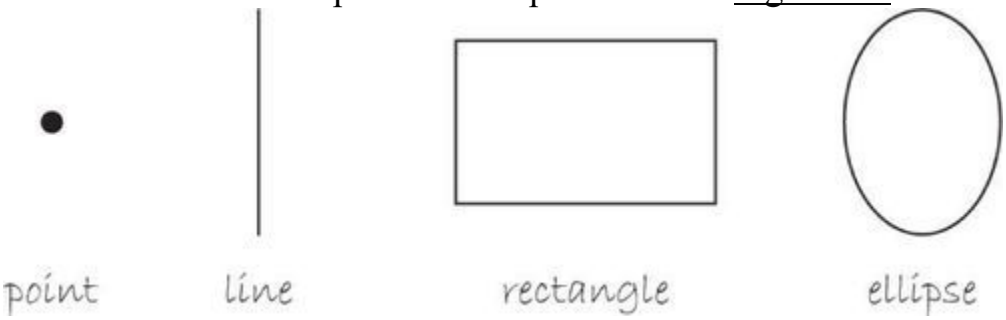


FIGURE 1-4

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.

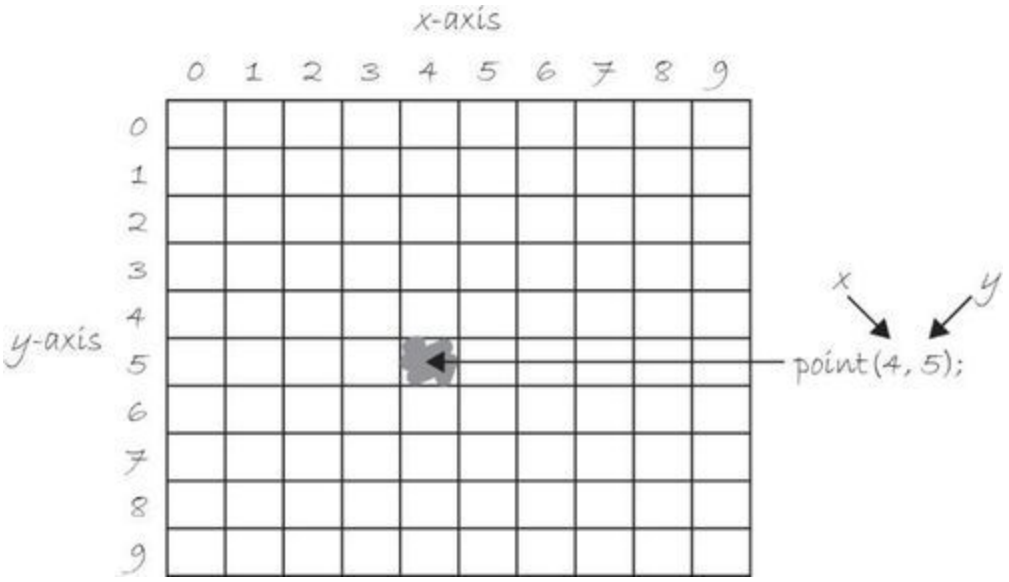


FIGURE 1-5

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.

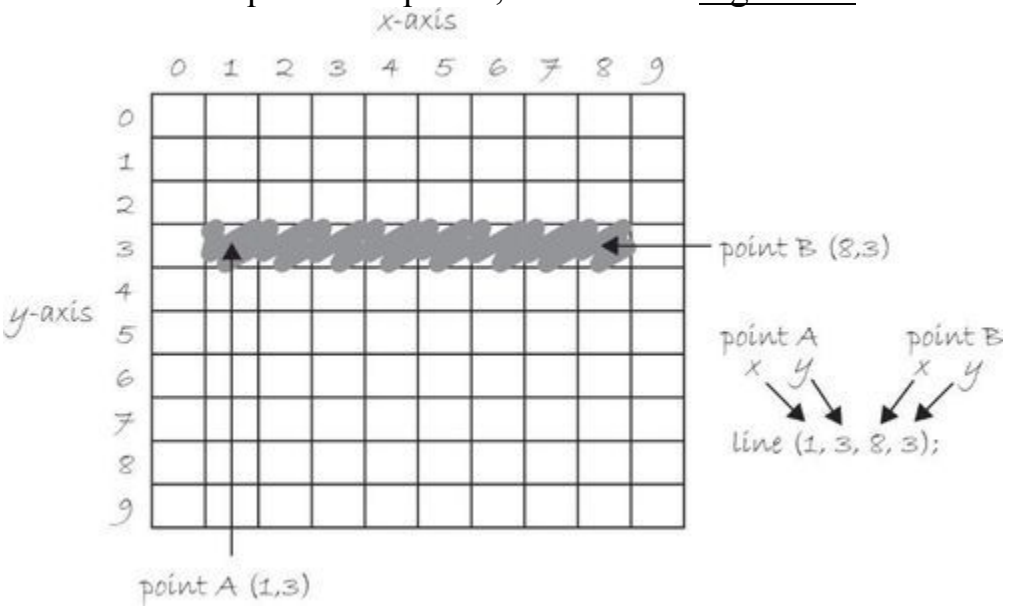


FIGURE 1-6

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).

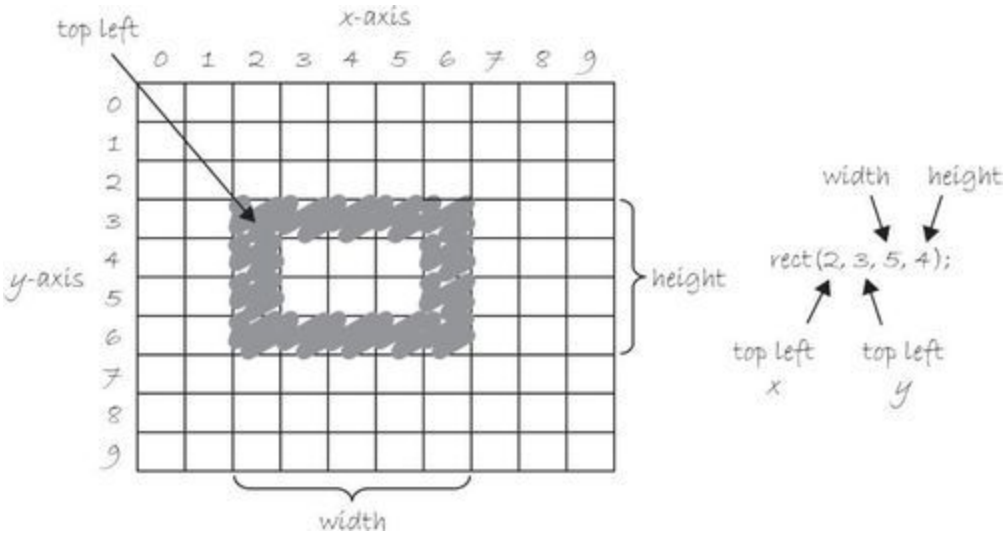


FIGURE 1-7

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.

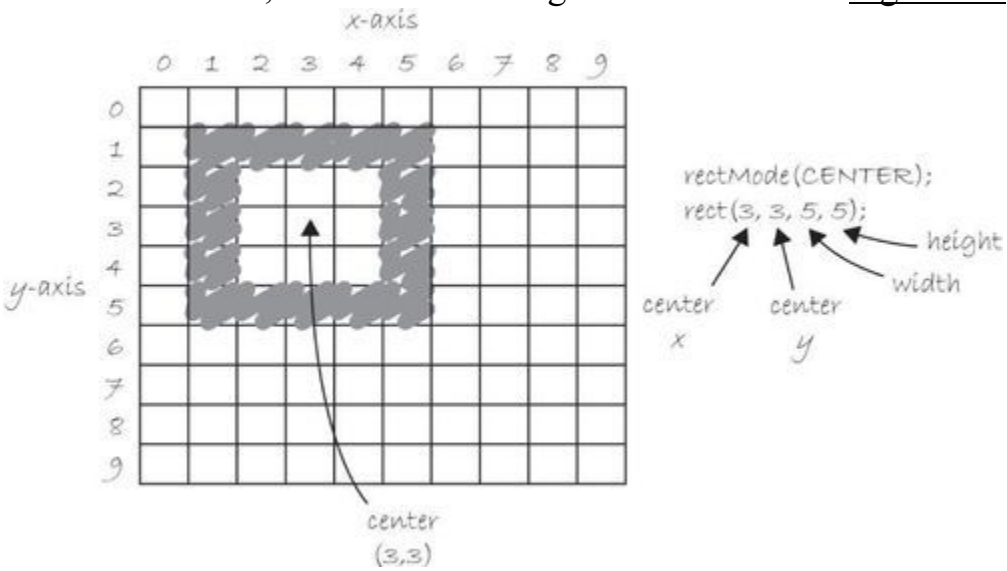


FIGURE 1-8

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).

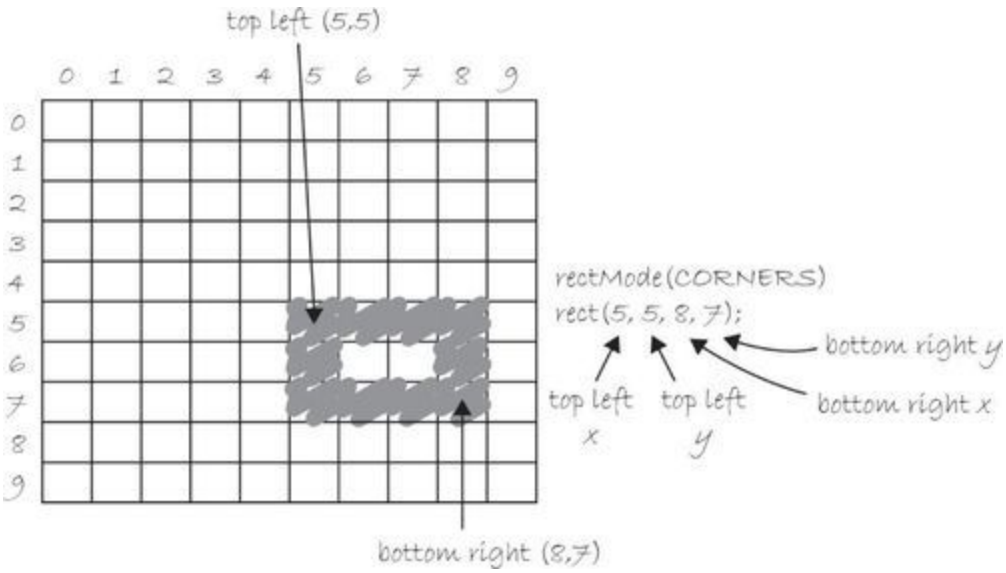


FIGURE 1-9

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.

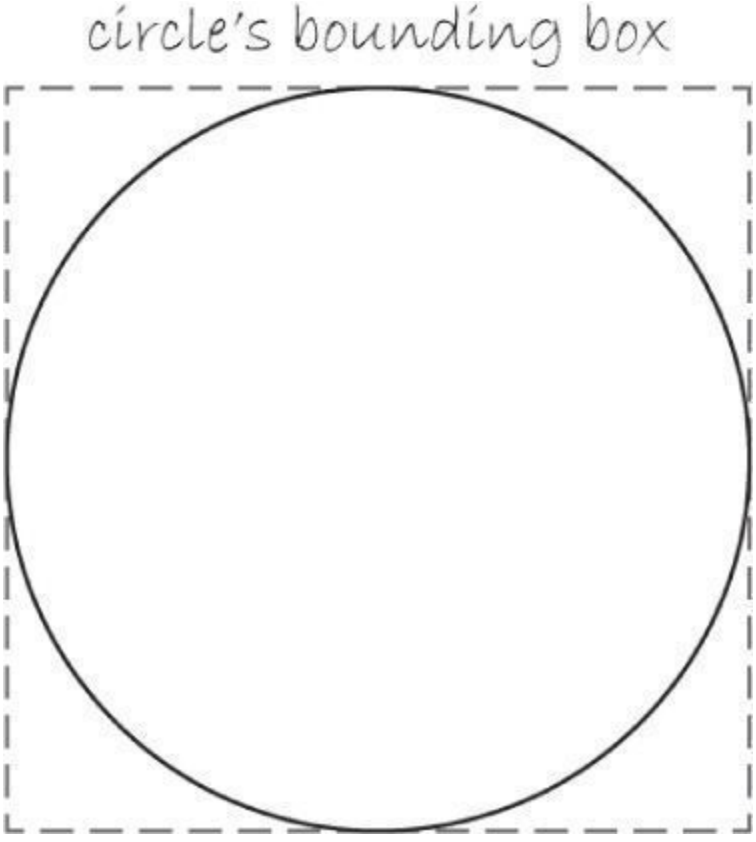


FIGURE 1-10

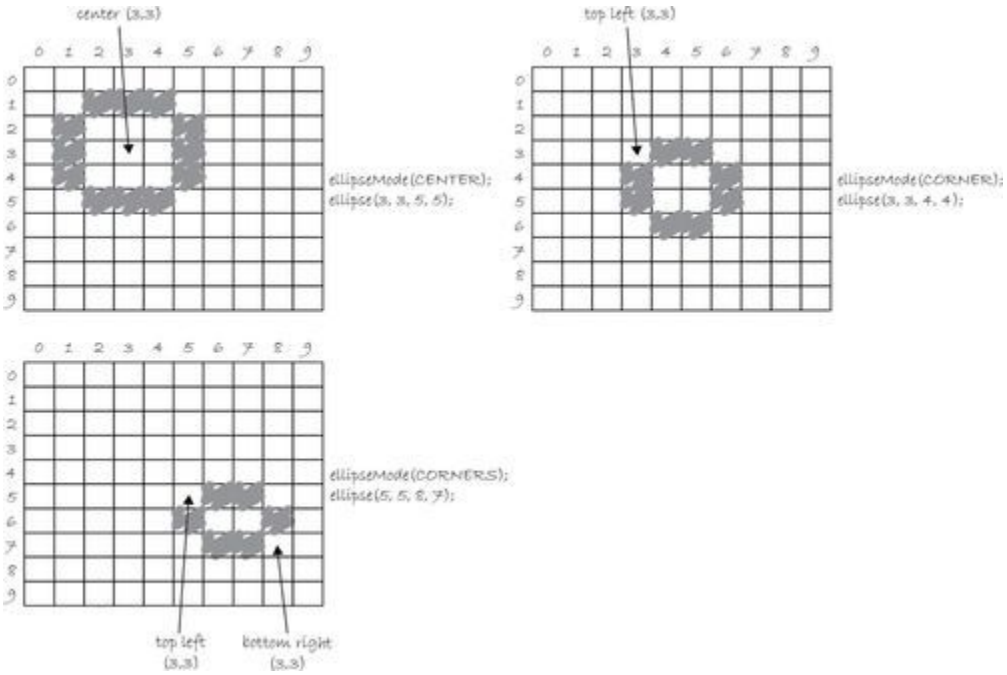


FIGURE 1-11

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()`.

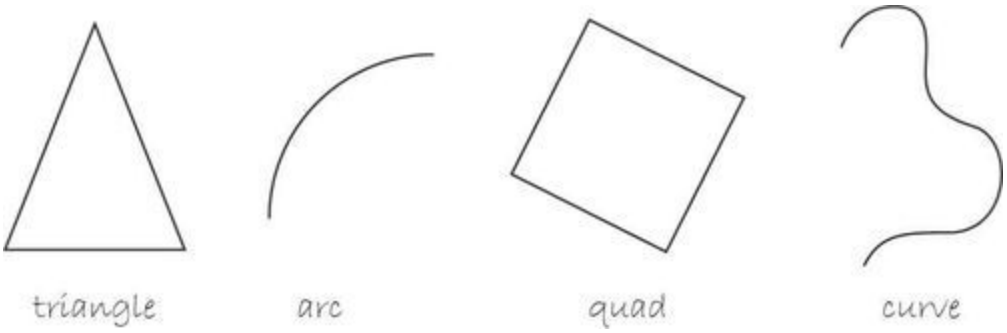
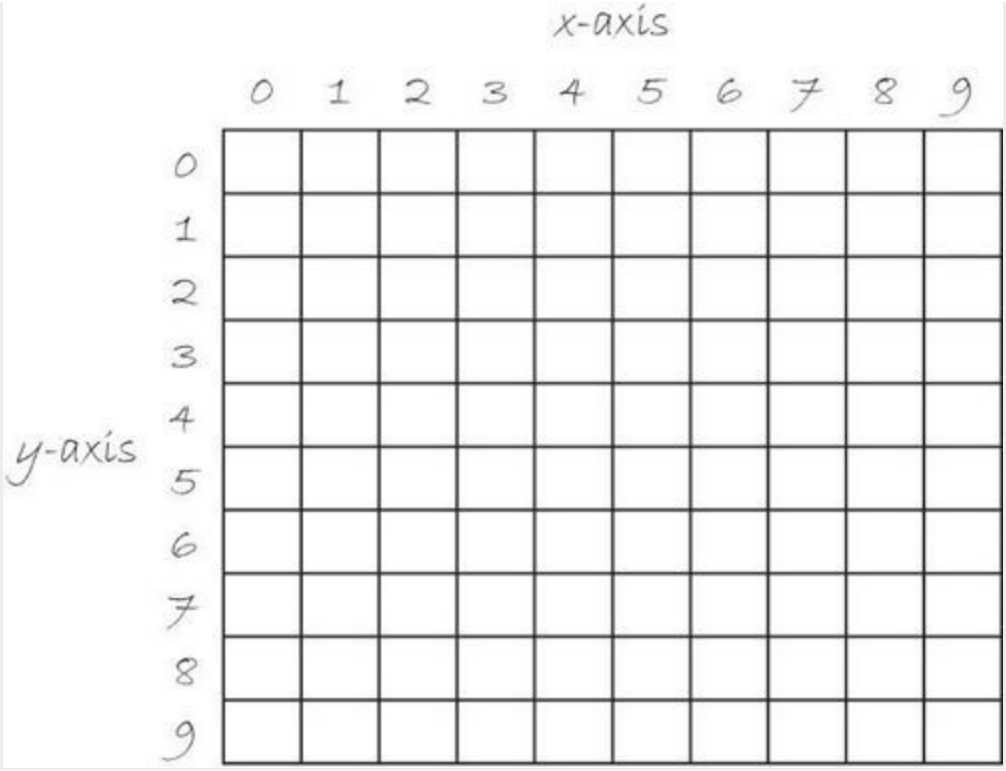


FIGURE 1-12

Exercise 1-2

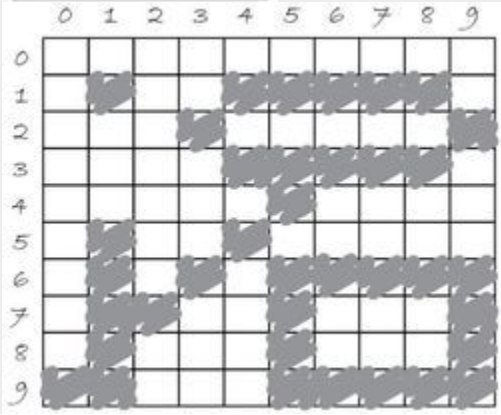
Using the blank graph below, draw the primitive shapes specified by the code.

```
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.

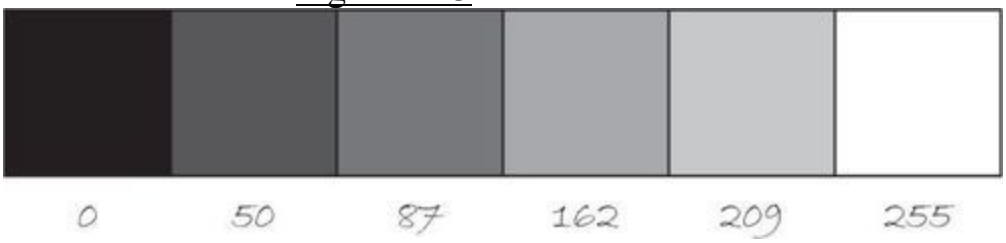


FIGURE 1-13

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.

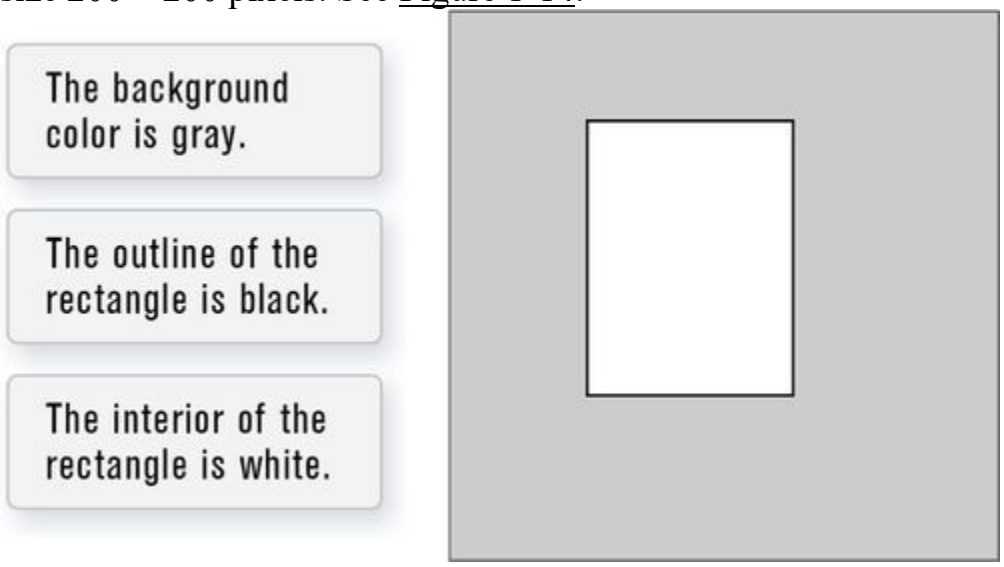


FIGURE 1-14

```
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.

Example 1-1

Stroke and fill

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

`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!

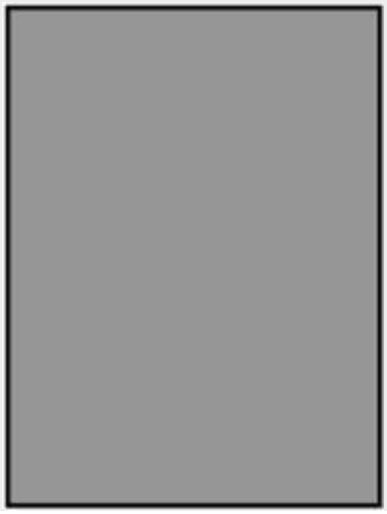


FIGURE 1-15

Example 1-2

noFill()

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

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);
```

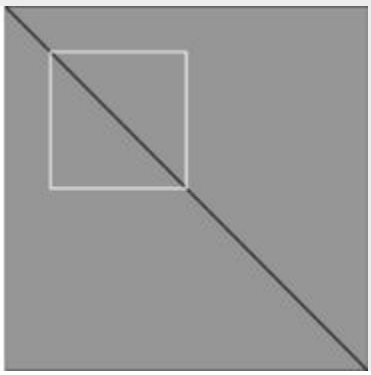


FIGURE 1-17

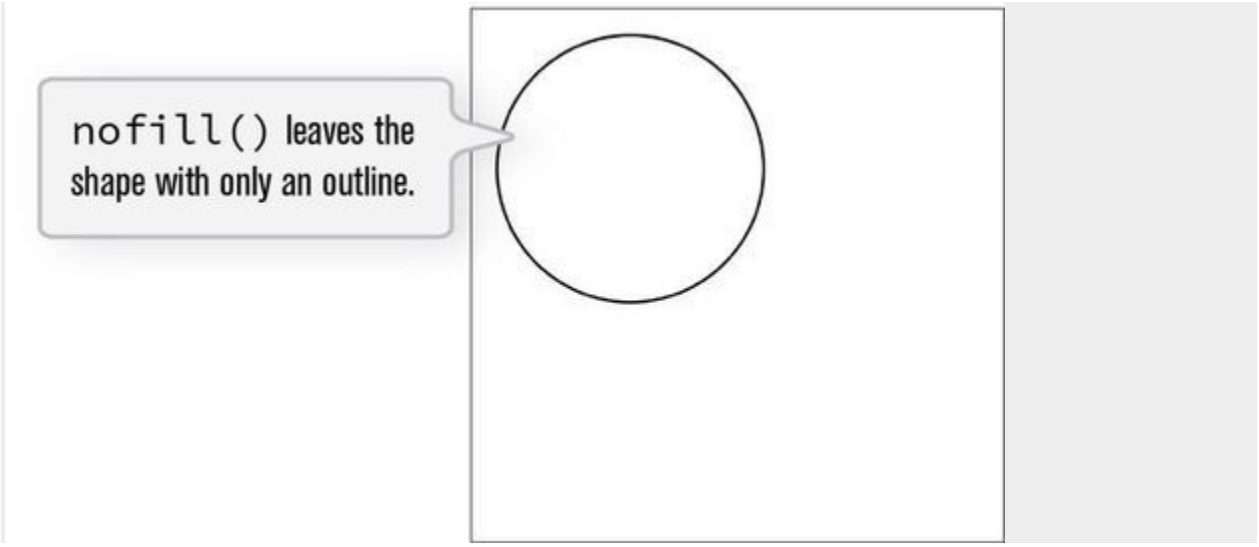


FIGURE 1-16

Exercise 1-4

Try to guess what the instructions would be for the following screenshot.

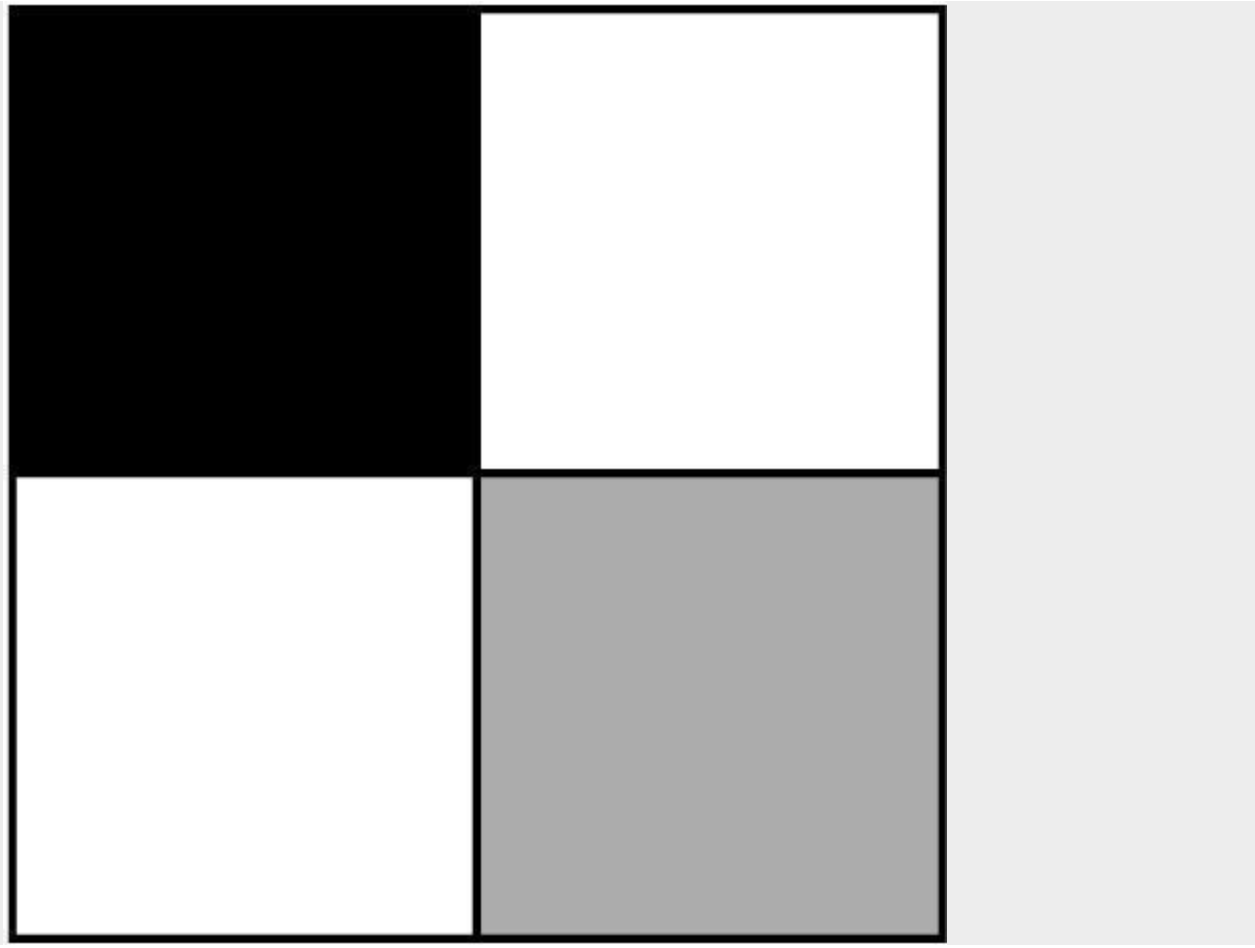
_____

_____

_____

_____

_____



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).

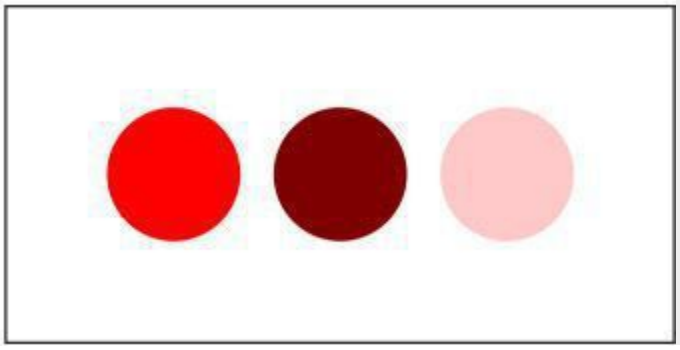


FIGURE 1-18

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.

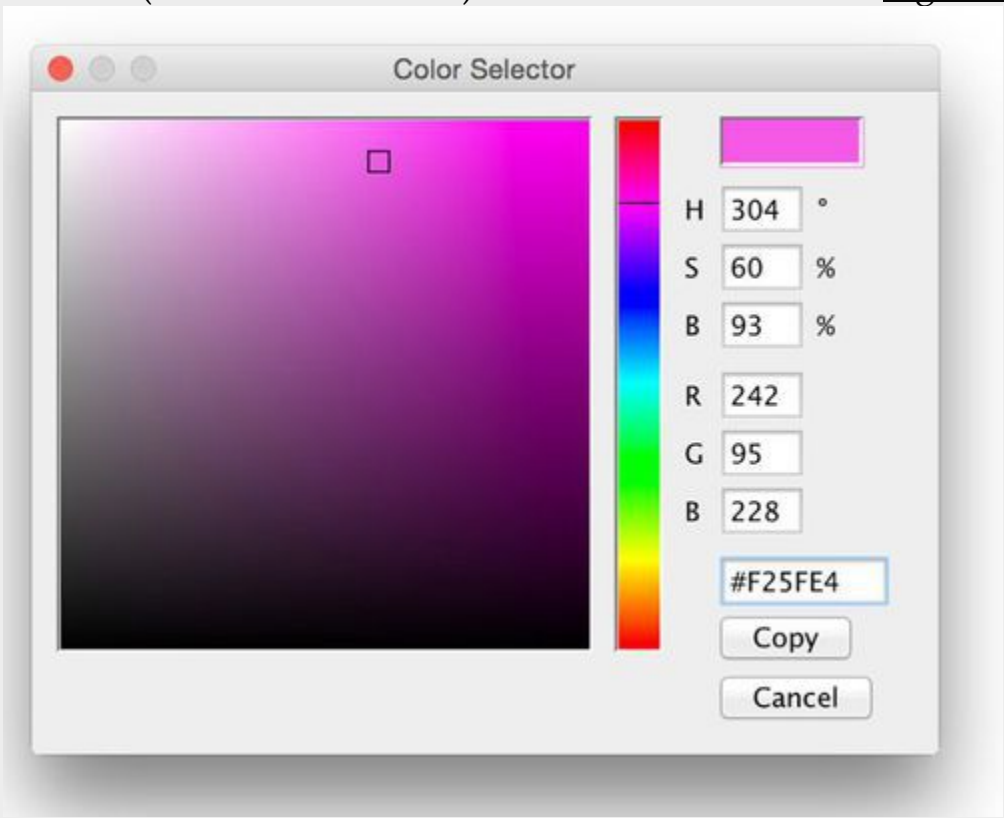


FIGURE 1-19

Exercise 1-5

Complete the following program. Guess what RGB values to use (you will be able to checkyour 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

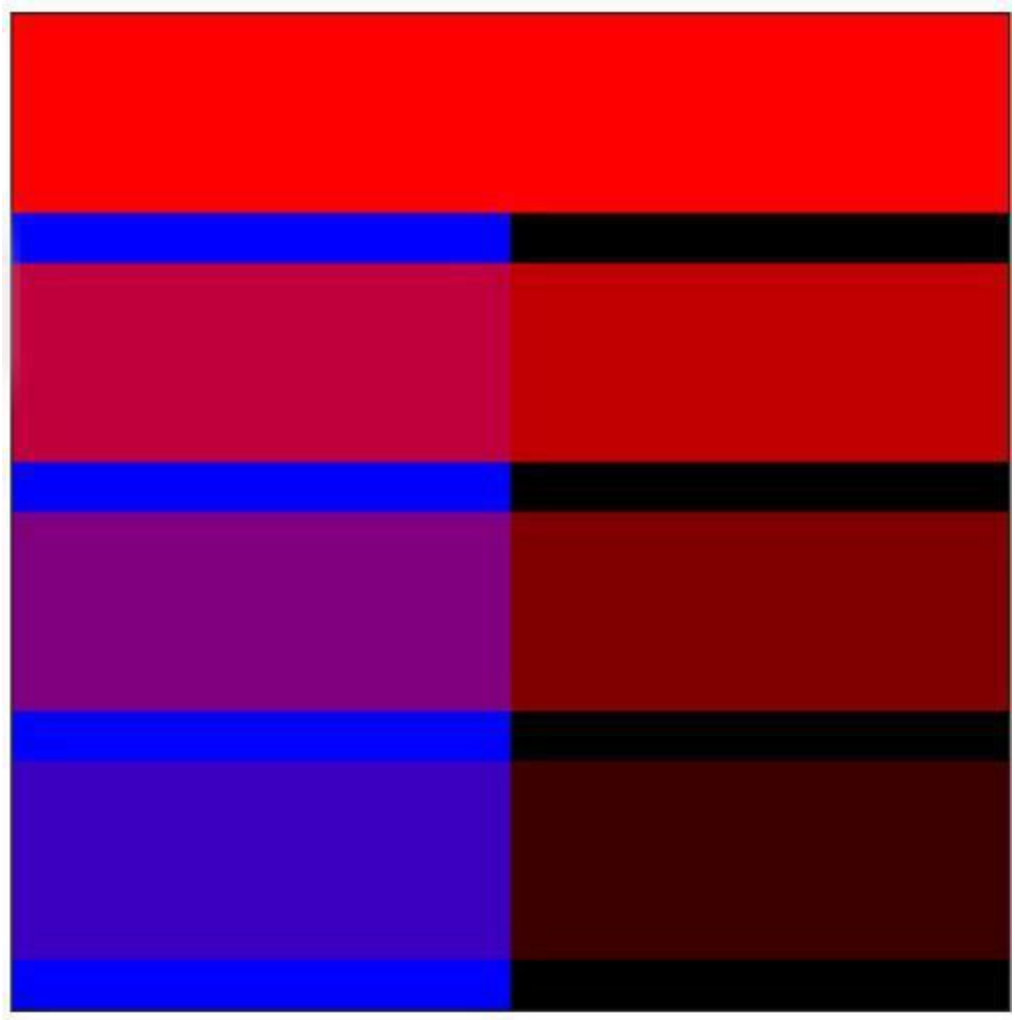


FIGURE 1-20

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

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.

[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.

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.)

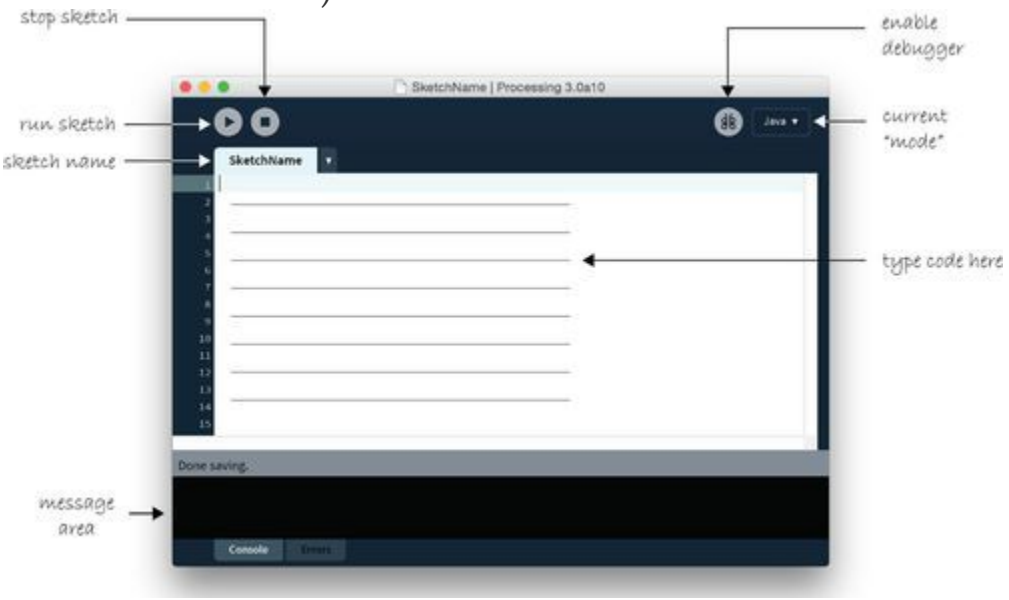


FIGURE 2-1

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.

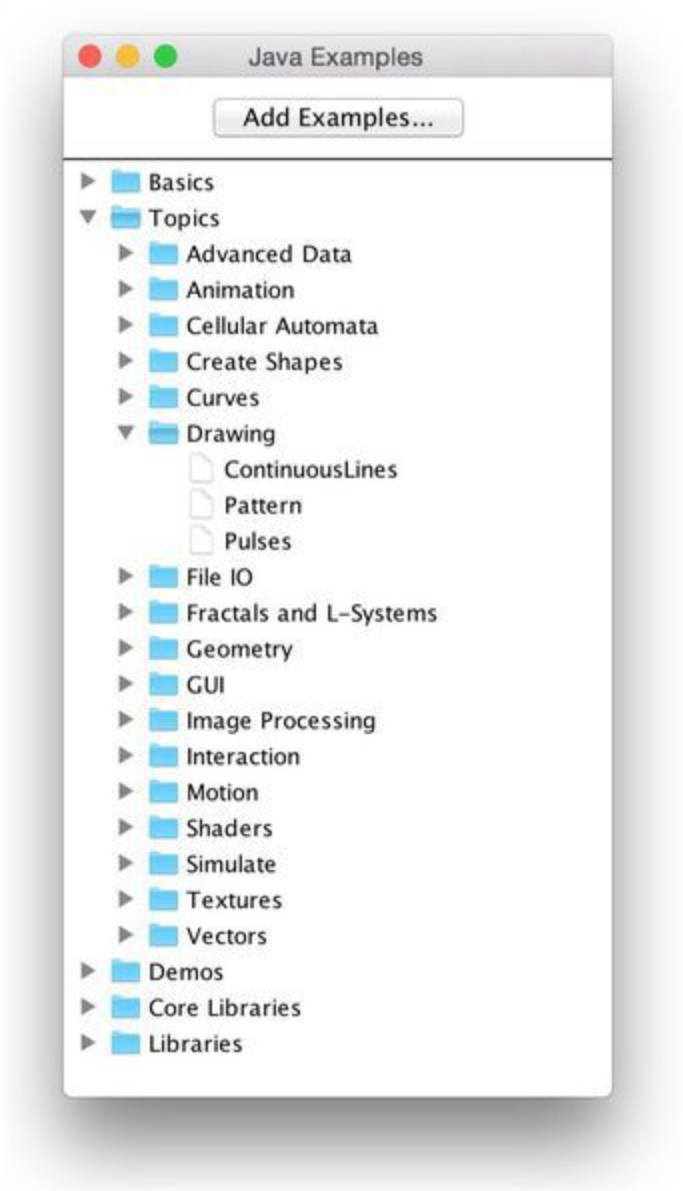


FIGURE 2-2

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.

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.

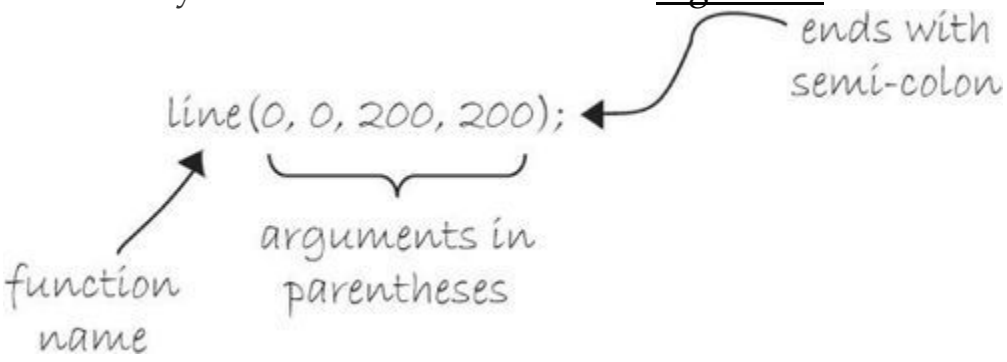


FIGURE 2-3

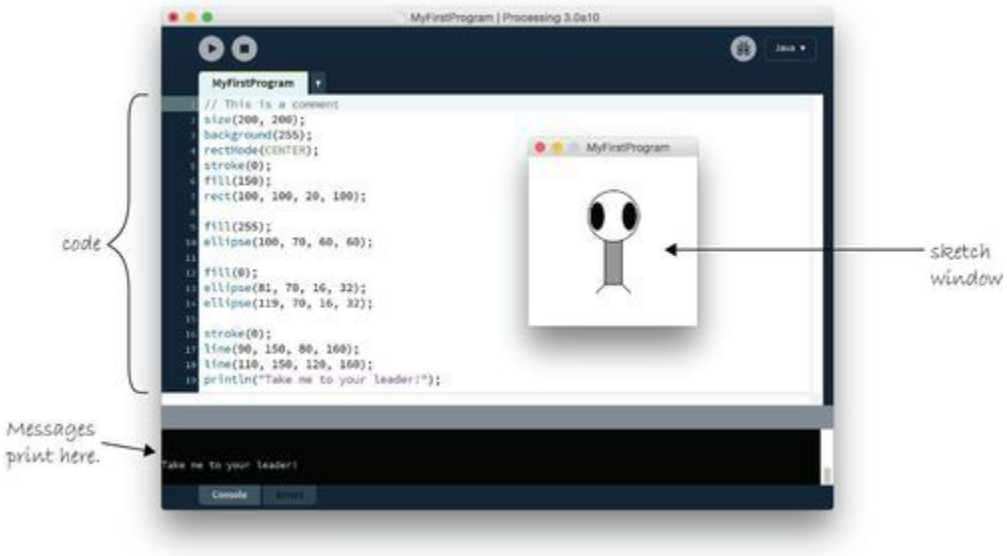


FIGURE 2-4

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).
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.

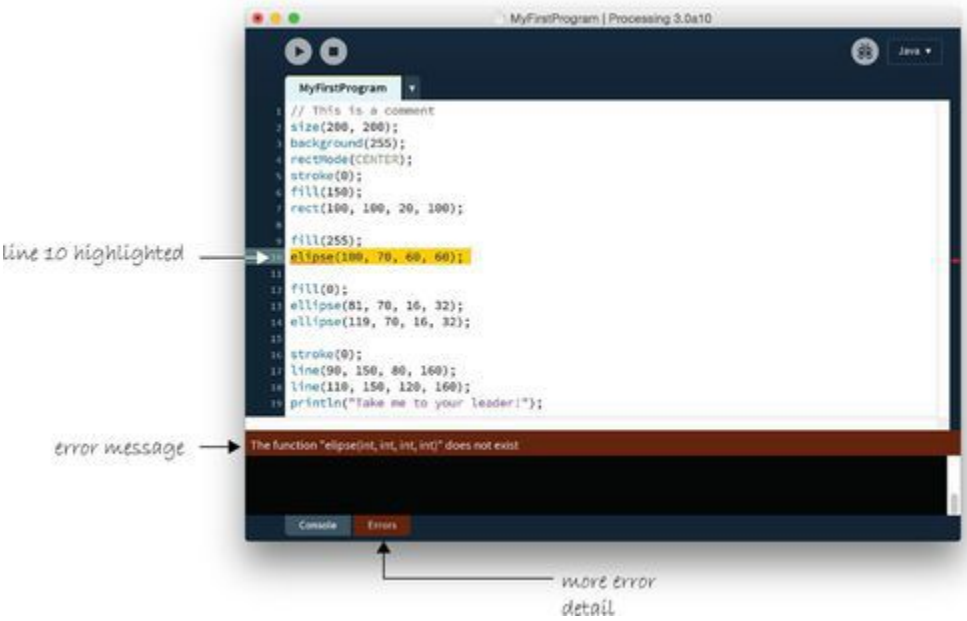


FIGURE 2-5

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.

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.

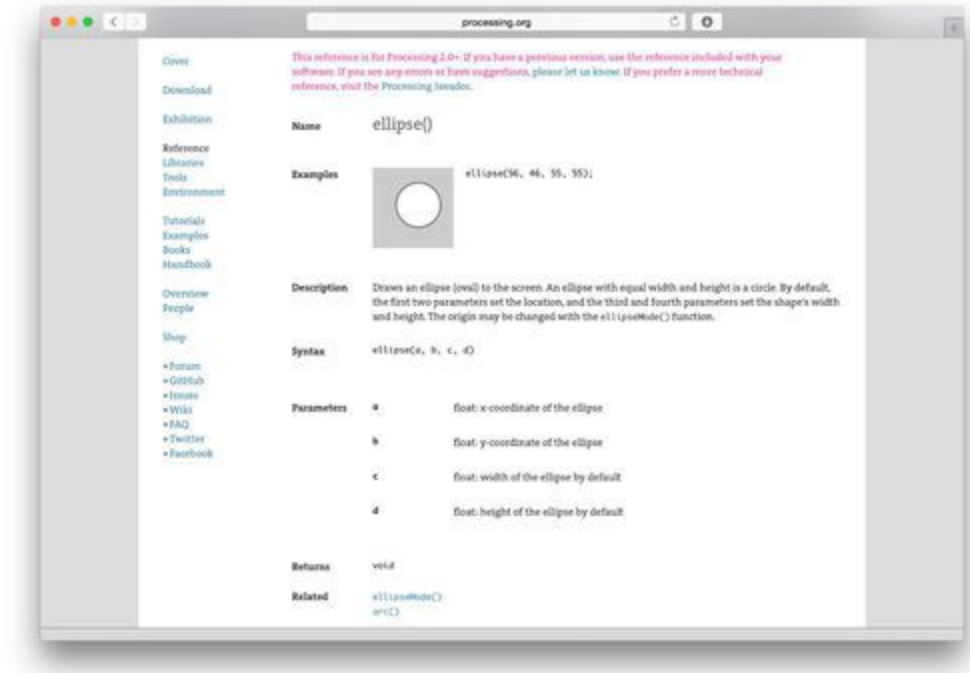


FIGURE 2-6

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.

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.

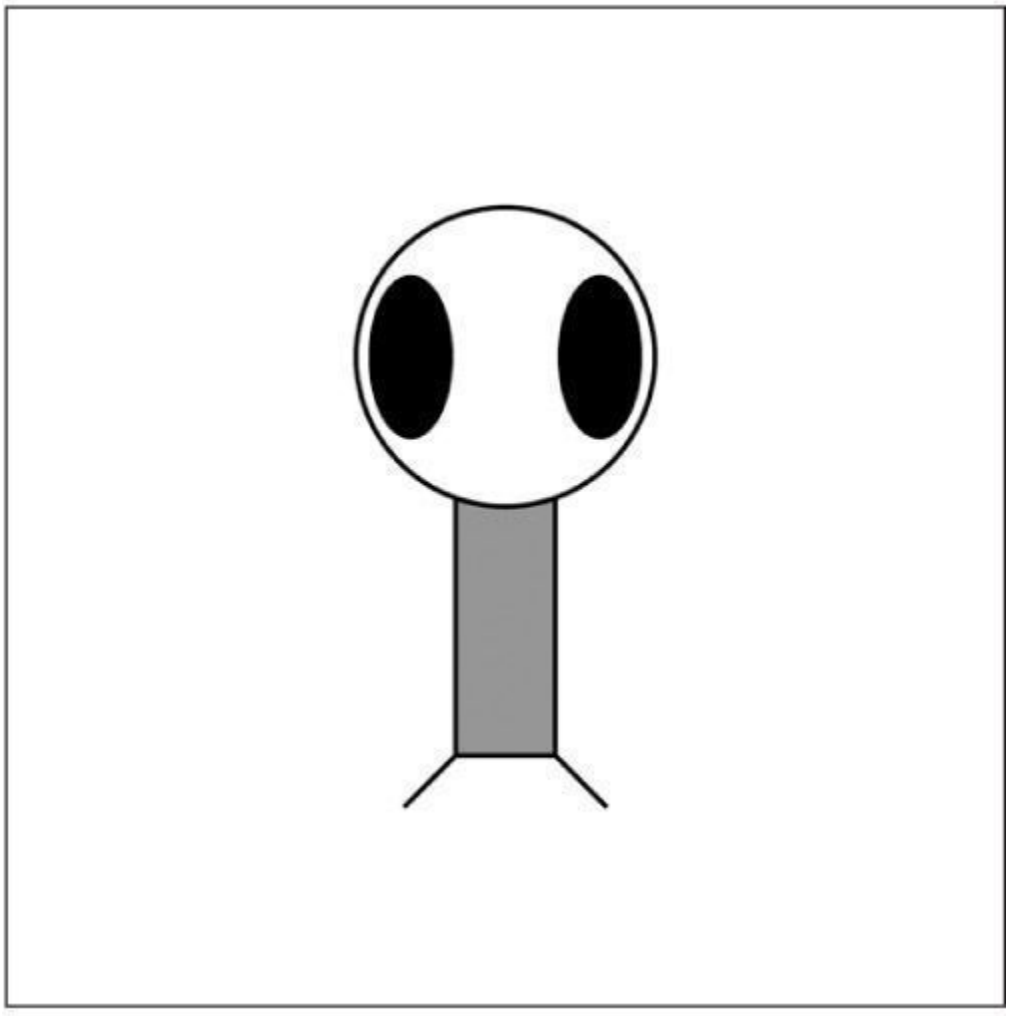


FIGURE 2-7

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);
```

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.

7

Functions

When it's all mixed up, better break it down.

—Tears for Fears

In this chapter:
– Modularity
– Declaring and defining a function
– Calling a function
– Arguments and parameters
– Returning a value
– Reusability

7-1 Break it down

The examples provided in Chapter 1 through Chapter 6 are short. I probably have not shown a sketch with more than 100 lines of code. These programs are the equivalent of writing the opening paragraph of this chapter, as opposed to the whole chapter itself.

Processing is great because you can make interesting visual sketches with small amounts of code. But as you move forward to looking at more complex projects, such as network applications or image processing programs, you will start to have hundreds of lines of code. You will be writing essays, not paragraphs. And these large amounts of code can prove to be unwieldy inside of your two main blocks — `setup()` and `draw()`.

Functions are a means of taking the parts of a program and separating them out into modular pieces, making code easier to read, as well as to revise. Let's consider the video game Space Invaders. The steps for `draw()` might look something like:

• Erase background.
• Draw spaceship.
• Draw enemies.

• Move spaceship according to user keyboard interaction.
• Move enemies.

What's in a name?

Functions are often called other things, such as "procedures" or "methods" or "subroutines." In some programming languages, there is a distinction between a procedure (performs a task) and a function (calculates a value). In this chapter, I am choosing to use the term function for simplicity's sake. Nevertheless, the technical term in the Java programming language is *method* (related to Java's object-oriented design) and once I get into objects in Chapter 8, I will use the term "method" to describe functions inside of objects.

Before this chapter on functions, I would have translated the above pseudocode into actual code, and placed it inside `draw()`. Functions, however, will let you approach the problem as follows:

```
void draw() {
  background(0);
  drawSpaceShip();
  drawEnemies();
  moveShip();
  moveEnemies();
}
```

I am calling functions I made up inside of `draw()`!

The above demonstrates how functions make life easier with clear and easy to manage code. Nevertheless, I am missing an important piece: the function *definitions*. Calling a function is old hat. You do this all the time when you write `line()`, `rect()`, `fill()`, and so on. Defining a new "made-up" function is going to require some more work on your part.

Before I launch into the details, let's reflect on why writing your own functions is so important:

• **Modularity** — Functions break down a larger program into smaller parts, making code more manageable and readable. Once I have figured out how to draw a spaceship, for example, I can take that chunk of spaceship drawing code, store it away in a function, and call upon that function whenever necessary (without having to worry about the details of the operation itself).

• **Reusability** — Functions allow you to reuse code without having to retype it. What if I want to make a two player Space Invaders game with two spaceships? I can `reuse` the `drawSpaceShip()` function by calling it multiple times without having to repeat code over and over.

In this chapter, I will look at some of my previous sketches, written without functions, and demonstrate the power of modularity and reusability by incorporating functions. In addition, I will further emphasize the distinctions between local and global variables, as functions are independent blocks of code that will require the use of local variables. Finally, I will continue to follow Zoog's story with functions.

Exercise 7-1

Write your answers below.

What functions might you write for your Lesson Two Project?	*What functions might you write in order t game Pong?*

7-2 "User-defined" functions

In Processing, you have been using functions all along. When you say `line(0, 0, 200, 200);` you are calling the function `line()`, a built-in function of the Processing environment. The ability to draw a line by calling the function `line()` does not magically exist. Someone, somewhere, defined (i.e., wrote the underlying code for) how Processing should display a line. One of Processing's strengths is its library of available functions, which you have started to explore throughout the first six chapters of this book. Now it's time to move beyond the built-in functions of Processing and write your own *user-defined*(a.k.a. "made-up") functions.

7-3 Defining a function

A function definition (sometimes referred to as a “declaration”) has three parts:
• Return type.
• Function name.
• Arguments.
It looks like this:

```
returnType functionName(parameters) {
// Code body of function
}
```

Déjà vu?

Remember when in Chapter 3 I introduced the functions `setup()` and `draw()`? Notice that they follow the same format you are learning now.
`setup()` and `draw()` are functions you define and are called automatically by Processing in order to run the sketch. All other functions you write have to be called by you.

For now, let’s focus solely on the *function name* and *code body*, ignoring *return type* and *parameters*.
Here is a simple example:

Example 7-1

Defining a function

```
void drawBlackCircle() {
fill(0);
ellipse(50, 50, 20, 20);
}
```

This is a simple function that performs one basic task: drawing an ellipse colored black at coordinate (50,50). Its name — `drawBlackCircle()` — is arbitrary (I made it up) and its code body consists of two instructions (you can have as much or as little code as you choose). It’s also important to remind ourselves that this is only the definition of the function. The code will never happen unless the function is actually called from a part of the program that is being executed. This is accomplished by referencing the function name, that is, calling a function, as shown in Example 7-2.

Example 7-2

Calling a function

```
void draw() {
background(255);
drawBlackCircle();
}
```

Exercise 7-2

Write a function that displays Zoog (or your own design). Call that function from within `draw()`.

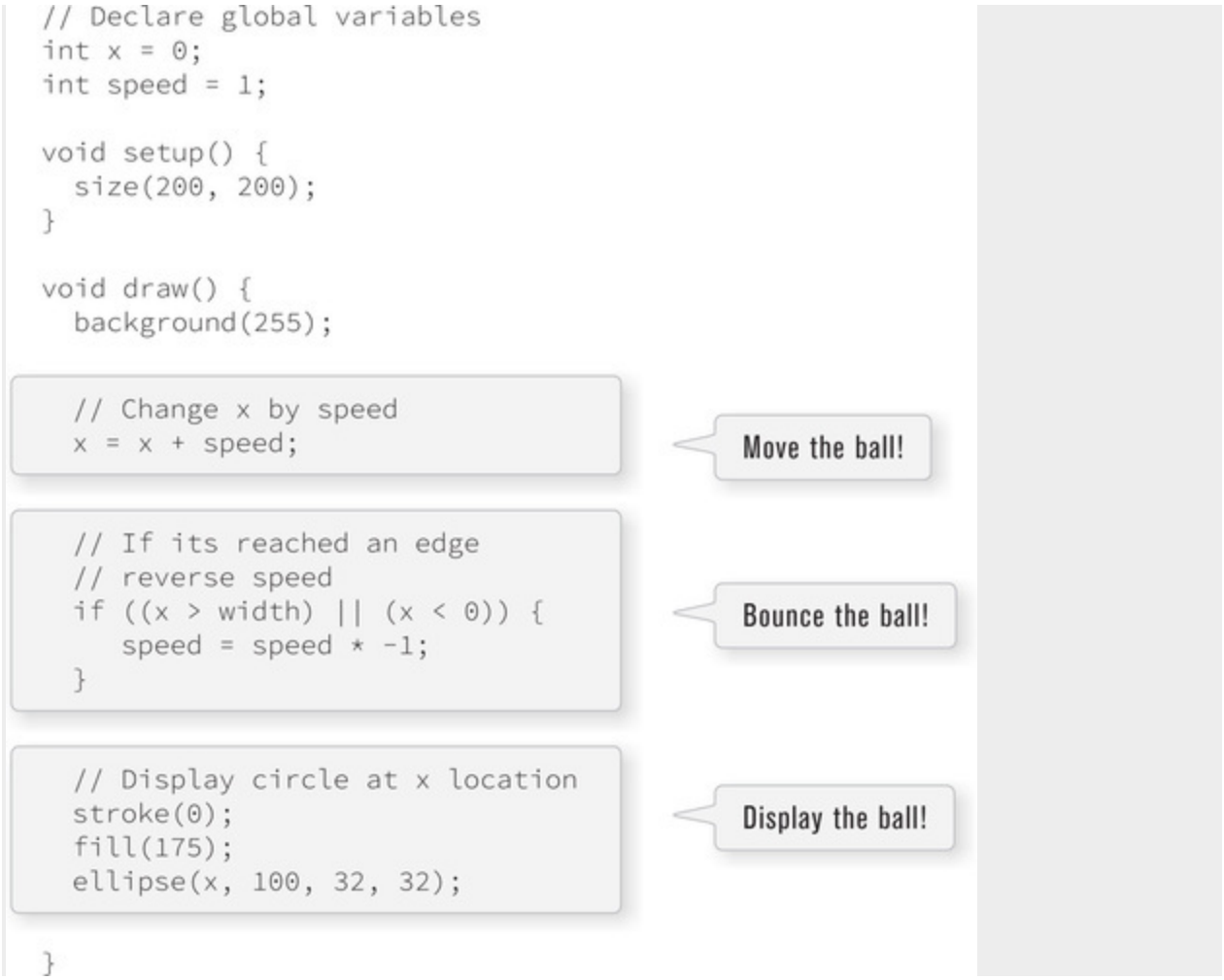
```
void setup() {
 size(200, 200);
}
void draw() {
 background(0);

 _____________________________________________________________
}

_____________________ ____________________ ___________________
{
_____________________________________________________________
_____________________________________________________________
_____________________________________________________________
_____________________________________________________________
____________________
```

7-4 Simple modularity

Let's examine the bouncing ball example from Chapter 5 and rewrite it using functions, illustrating one technique for breaking a program down into modular parts. Example 5-6 is reprinted here for your convenience.

Example 7-3

Bouncing ball



Once I have determined how I want to divide the code up into functions, I can take the pieces out of `draw()` and insert them into function definitions, calling those functions inside `draw()`. Functions typically are written below `draw()`.

Example 7-4

Bouncing ball with functions

```
// Declare all global variables (stays the same)
int x = 0;
int speed = 1;

// Setup does not change
void setup() {
  size(200, 200);
}

void draw() {
  background(255);
  move();
  bounce();
  display();
}

// A function to move the ball
void move() {
  // Change the x location by speed
  x = x + speed;
}

// A function to bounce the ball
void bounce() {
  // If its reached an edge, reverse speed
  if ((x > width) || (x < 0)) {
    speed = speed * -1;
  }
}

// A function to display the ball
void display() {
  stroke(0);
  fill(175);
  ellipse(x, 100, 32, 32);
}
```

Instead of writing out all the code about the ball in `draw()`, I simply call three functions. How do I know the names of these functions? I made them up!

Where should functions be placed? You can define your functions anywhere in the code outside of `setup()` and `draw()`. However, the convention is to place your function definitions below `draw()`.

Note how simple `draw()` has become. The code is reduced to function *calls*; the detail for how variables change and shapes are displayed is left for the function *definitions*. One of the main benefits here is the programmer's sanity. If you wrote this program right before leaving on a two-week vacation in the Caribbean, upon returning with a nice tan, you would be greeted by well-organized, readable code. To change how the ball is rendered, you only need to make edits to the `display()` function, without having to search through long passages of code or worrying about the rest of the program. For example, try replacing `display()` with the following:

```
void display() {
  background(255);
  rectMode(CENTER);
  noFill();
  stroke(0);
  rect(x, y, 32, 32);
  fill(255);
    rect(x-4, y-4, 4, 4);
    rect(x+4, y-4, 4, 4);
    line(x-4, y+4, x+4, y+4);
}
```

If you want to change the appearance of the shape, the `display()` function can be rewritten leaving all the other features of the sketch intact.

Another benefit of using functions is greater ease in debugging. Suppose, for a moment, that the bouncing ball function was not behaving appropriately. In order to find the problem, I now have the option of turning on and off parts of the program. For example, I might simply run the program with `display()` only, by commenting out `move()` and `bounce()`:

```
void draw() {
  background(0);
  // move();
  // bounce();
  display();
}
```

Functions can be commented out to determine if they are causing a bug or not.

The function definitions for `move()` and `bounce()` still exist, only now the functions are not being called. By adding function calls one by one and executing the sketch each time, I can more easily find the location of the problematic code.

Exercise 7-3

Take any Processing program you have written and modularize it using functions, as above. Use the following space to make a list of functions you need to write.

_____
_____
_____
_____
_____

7-5 Arguments

Just a few pages ago I said "Let's ignore **ReturnType** and **Arguments**." I did this in order to ease into functions by sticking with the basics. However, functions possess greater powers than simply breaking a program into parts. One of the keys to unlocking these powers are the concepts of *arguments* and *parameters*.

Arguments are values that are "passed" into a function. You can think of them as inputs that a function needs to do its job. A function that causes a creature to move a certain number of steps needs to know how many steps you want the creature to move. Instead of merely saying "move," you might say, "move ten steps," where "ten" is the argument.

When you define such a "move" function, you are required to give each argument a name. That way, the function can refer to the arguments it receives by the particular name that you specify. To illustrate, let's rewrite `drawBlackCircle()` to include a parameter:

```
void drawBlackCircle(int diameter) {
  fill(0);
  ellipse(50, 50, diameter, diameter);
}
```

`diameter` is a parameter to the function `drawBlackCircle()`.

A parameter is simply a variable declaration inside the parentheses in the function definition. This variable is a *local variable* (remember the discussion in Section 6-5 on page 104?) to be used in that function (and only in that function). The black circle will be sized according to the value of diameter, which will automatically be assigned the value that you pass the function when you call it. For example, when you say `drawBlackCircle(100)`, the value `100` is the argument.

That `100` gets assigned to the `diameter` parameter, and the function itself

uses `diameter` to draw the circle. When you call `drawBlackCircle(80)`, the argument `80` is assigned to parameter `diameter`, and the function body then uses `diameter` to draw the circle.

```
drawBlackCircle(16); // Draw the circle with a diameter
of 16
drawBlackCircle(32); // Draw the circle with a diameter
of 32
```

You could also pass another variable or the result of a mathematical expression (such as `mouseX` divided by 10) into the function. For example:

```
drawBlackCircle(mouseX / 10);
```

This, by the way, is exactly what you did in Chapter 1 when you first started drawing in Processing. To draw a line, for example, you couldn't just say draw a line. Rather, you had to say draw a line from some (x,y) to some other (x,y). You needed *four arguments*.

```
line(10, 25, 100, 75); // Draw a line from (10,25) to
(100,75).
```

The key difference here is that you didn't write the `line()` function! The creators of Processing did, and if you delve into the Processing source itself, you'll find a function definition with *four parameters*.

```
void line(float x1, float y1, float x2, float y2) {
 // This functions requires four parameters
 // which define the end points (x1,y1) and (x2,y2)
 // of a line!
}
```

Parameters pave the way for more flexible, and therefore reusable, functions. To demonstrate this, let's look at code for drawing a collection of shapes and examine how functions allow you to draw multiple versions of the pattern without retyping the same code over and over.

Leaving Zoog until a bit later, consider the following pattern resembling a car (viewed from above as shown in Figure 7-1):

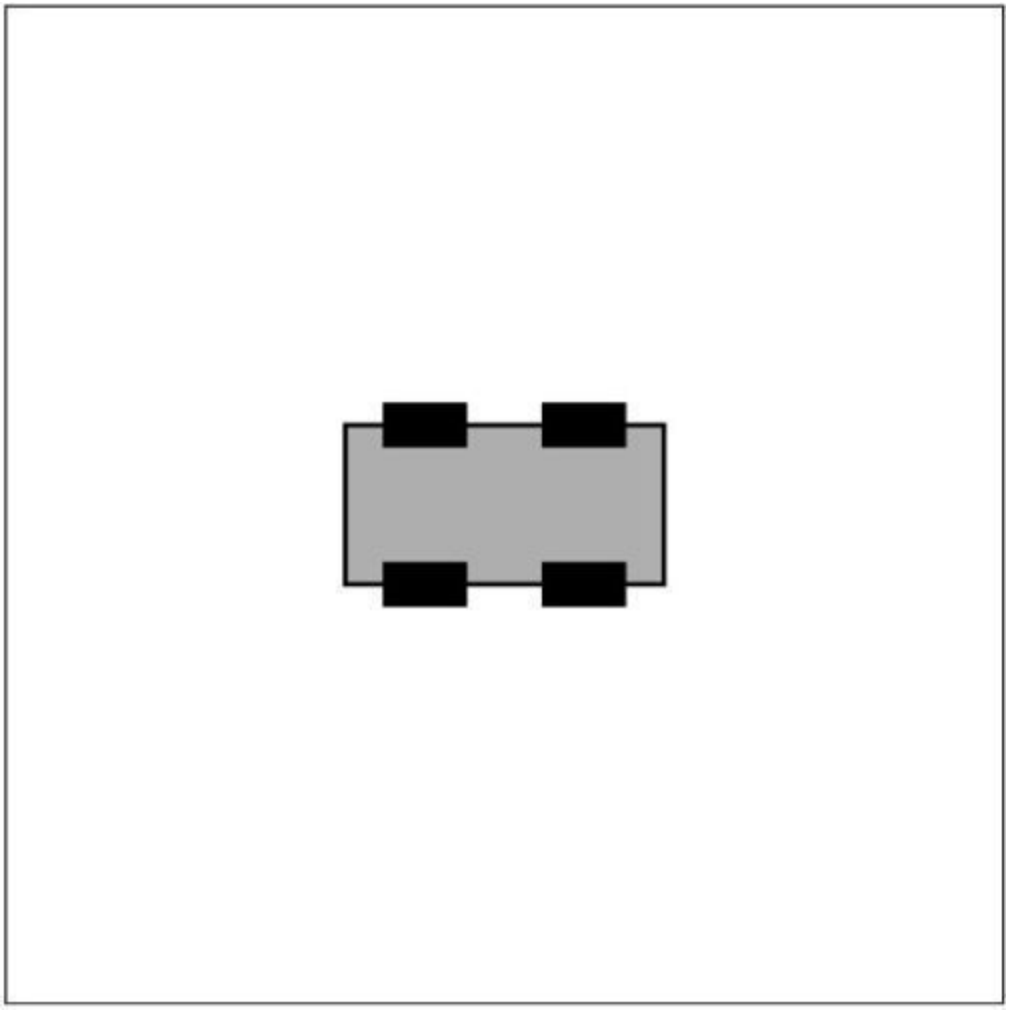


FIGURE 7-1

```
size(200, 200);
background(255);
int x = 100;                // x location
int y = 100;                // y location
int thesize = 64;           // size
int offset = thesize/4;     // position of wheels
relative to car

// Draw main car body (i.e., a rect)
rectMode(CENTER);
stroke(0);
fill(175);
rect(x, y, thesize, thesize/2);

// Draw four wheels relative to center
fill(0);
rect(x - offset, y - offset, offset, offset/2);
rect(x + offset, y - offset, offset, offset/2);
rect(x - offset, y + offset, offset, offset/2);
rect(x + offset, y + offset, offset, offset/2);
```

The car shape is five rectangles, one large rectangle in the center...

...and four wheels on the outside.

To draw a second car, I'll repeat the above code with different values, as shown in Figure 7-2.

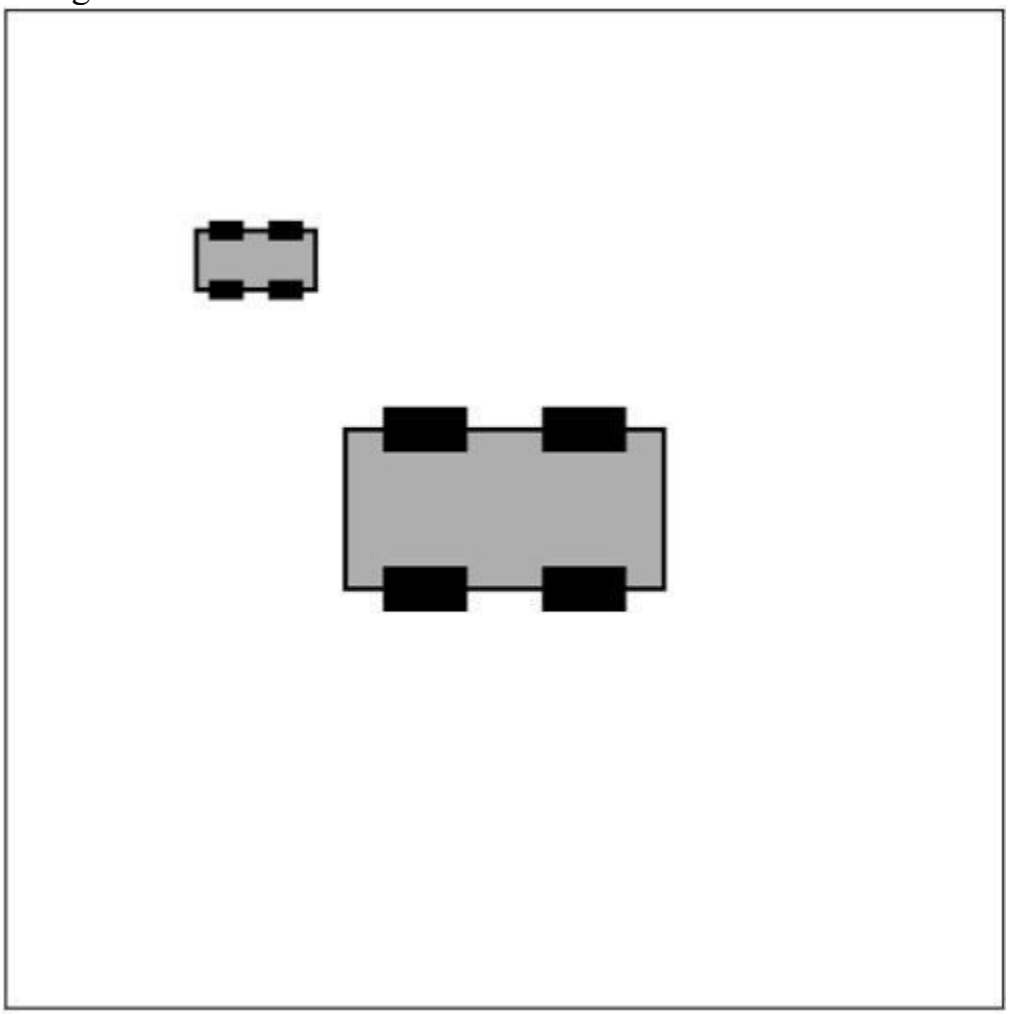


FIGURE 7-2

```
x = 50;                 // x location
y = 50;                 // y location
thesize = 24;           // size
offset = thesize/4;     // position of wheels relative
to car

// Draw main car body (i.e., a rect)
rectMode(CENTER);
stroke(0);
fill(175);
rect(x, y, thesize, thesize/2);

// Draw four wheels relative to center
fill(0);
rect(x - offset, y - offset, offset, offset/2);
rect(x + offset, y - offset, offset, offset/2);
rect(x - offset, y + offset, offset, offset/2);
rect(x + offset, y + offset, offset, offset/2);
```

Every single line of code is repeated to draw the second car.

It should be fairly apparent where this is going. After all, I am doing the same thing twice — why bother repeating all that code? To escape this repetition, I can move the code into a function that displays the car according to several parameters (position, size, and color).

```
void drawCar(int x, int y, int theSize, color c) {
  // Using a local variable "offset "
  int offset = theSize/4;
  // Draw main car body
  rectMode(CENTER);
  stroke(200);
  fill(c);
  // Draw four wheels relative to center
  fill(200);
  rect(x - offset, y - offset, offset, offset/2);
  rect(x + offset, y - offset, offset, offset/2);
  rect(x - offset, y + offset, offset, offset/2);
  rect(x + offset, y + offset, offset, offset/2);
}
```

Local variables can be declared and used in a function!

This code is the **function definition**. The function `drawCar()` draws a car shape based on four arguments: horizontal location, vertical location, size, and color.

In the `draw()` function, I then call the `drawCar()` function three times, passing four *arguments* each time. See the output in Figure 7-3.

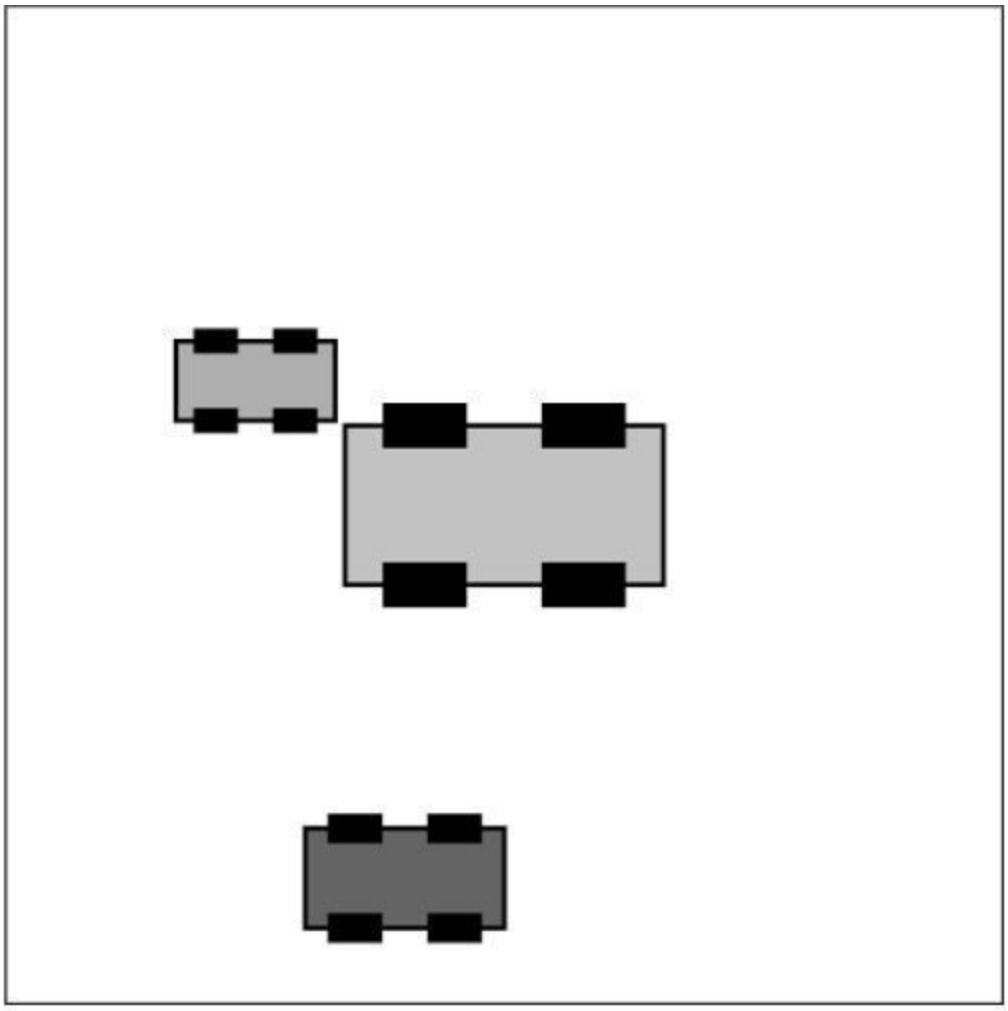


FIGURE 7-3

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

void draw() {
  background(255);
  drawCar(100, 100, 64, color(200, 200, 0));
  drawCar(50, 75, 32, color(0, 200, 100));
  drawCar(80, 175, 40, color(200, 0, 0));
}
```

This code calls the function three times, with the exact number of parameters in the right order.

Technically speaking, *parameters* are the variables that live inside the parentheses in the function definition, that is, `void drawCar(int x, int y, int thesize, color c)`. *Arguments* are the values passed into the function when

it is called, that is, `drawCar(80, 175, 40, color(100, 0, 100))`. The semantic difference between arguments and parameters is somewhat trivial and you should not be terribly concerned if you confuse the use of the two words from time to time.

The concept to focus on is this ability to *pass* arguments. You will not be able to advance your programming knowledge unless you are comfortable with this technique.

Let's go with the word *pass*. Imagine a lovely, sunny day and you're playing catch with a friend in the park. You have the ball. You (the main program) call the function (your friend) and pass the ball (the argument). Your friend (the function) now has the ball (the argument) and can use it however he or she pleases (the code itself inside the function). See Figure 7-4.

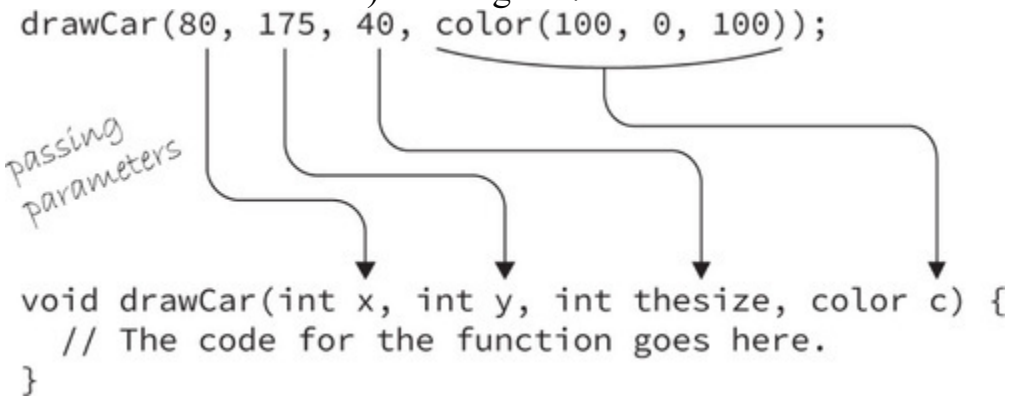


FIGURE 7-4

Important things to remember about passing arguments

- You must pass the same number of arguments as defined in the function's parameters.
- When an argument is passed, it must be of the same *type* as declared within the parameters in the function definition. An integer must be passed into an integer, a floating point into a floating point, and so on.
- The value you pass to a function can be a literal value (`20`, `5`, `4.3`, etc.), a variable (`x`, `y`, etc.), or the result of an expression (`8 + 3`, `4 * x/2`, `random(0, 10)`, etc.)
- Parameters act as local variables to a function and are only accessible within that function.

Exercise 7-4

The following function takes three numbers, adds them together, and prints the sum to the message window.

```
void sum(int a, int b, int c) {
int total = a + b + c;
println(total);
}
```

Looking at the function definition above, write the code that calls the function.

_____

Exercise 7-5

OK, here is the opposite problem. Here is a line of code that assumes a function that takes two numbers, multiplies them together, and prints the result to a message window. Write the function definition that goes with this function call.

```
multiply(5.2, 9.0);
```

_____

_____

_____

_____

 *Exercise 7-6: Create a design for a flower. Can you write a function with parameters that vary the flower's appearance (height, color, number of petals, etc.)? If you call that function multiple times with different arguments, can you create a garden with a variety of flowers?*

7-6 Passing a copy

There is a slight problem with the "playing catch" analogy. What I really should have said is the following. Before tossing the ball (the argument), you make a copy of it (a second ball), and pass it to the receiver (the function).

Whenever you pass a primitive value (`int`, `float`, `char`, etc.) to a function, you do not actually pass the value itself, but a copy of that variable. This may seem like a trivial distinction when passing a hard-coded number, but it's not so trivial when passing a variable.

The following code has a function entitled `randomizer()` that receives one parameter (a floating point number) and adds a random number between −2 and 2 to it. Here is the pseudocode.

• **num** is the number 10.
• **num** is displayed: **10**
• **A copy of num** is passed into the parameter **newnum** in the function `randomizer()`.
• In the function `randomizer()`:
– a random number is added to **newnum**.
– **newnum** is displayed: **10.34232**
• **num** is displayed again: **Still 10! A copy was sent into newnum, so num has not changed**.

And here is the code:

```
void setup() {
float num = 10;
println("The number is: " + num);
randomizer(num);
println("The number is: " + num);
}
```

```
void randomizer(float newnum) {
newnum = newnum + random(-2, 2);
println("The new number is: " + newnum);
}
```

Even though the variable **num** was passed into the variable **newnum**, which then quickly changed values, the original value of the variable **num** was not affected because a copy was made.

I like to refer to this process as "pass by copy," however, it's more commonly referred to as "pass by value." This holds true for all primitive data types (the only kinds I've covered so far: integer, float, etc.), but is not quite the same with *objects*, which you will learn about in the next chapter.

This example also provides a nice opportunity to review the *flow* of a program when using a function. Notice how the code is executed in the order that the lines are written, but when a function is called, the code leaves its current line, executes the lines inside of the function, and then comes back to where it left off. Here is a description of the preceding example's flow:

1. Set num equal to 10.
2. Print the value of num.
3. Call the function randomizer.

a. Set newnum equal to newnum plus a random number.

b. Print the value of newnum.

4. Print the value of num.

Exercise 7-7

Predict the output of this program by writing out what would appear in the message window.

```
void setup() {
  println("a");
  function1();
  println("b");
}

void draw() {
  println("c");
  function2();
  println("d");
  function1();
  noLoop();
}

void function1() {
  println("e");
  println("f");
}

void function2() {
  println("g");
  function1();
  println("h");
}
```

New! `noLoop()` is a built-in function in Processing that stops `draw()` from looping. In this case, I can use it to ensure that `draw()` only executes one time. I could restart it at some other point in the code by calling the function `loop()`.

It's perfectly reasonable to call a function from within a function. In fact, you do this all the time whenever you call any function from inside of `setup()` or `draw()`.

Output:

```
1: _____   7: _____

2: _____   8: _____

3: _____   9: _____

4: _____  10: _____

5: _____  11: _____

6: _____  12: _____
```

7-7 Return type

So far you have seen how functions can separate a sketch into smaller parts, as well as incorporate arguments to make it reusable. However, there is one piece still missing from this discussion, and it is the answer to the question you have been wondering all along: "What does *void* mean?"

As a reminder, let's examine the structure of a function definition again:

```
returnType functionName(parameters) {
// Code body of function
}
```

OK, now let's look at one of the functions:

```
void drawCar(int x, int y, int theSize, color c) {
```

```
int offset = theSize/4;
// Draw main car body
rectMode(CENTER);
stroke(200);
fill(c);
// Draw four wheels relative to center
fill(200);
rect(x - offset, y - offset, offset, offset/2);
rect(x + offset, y - offset, offset, offset/2);
rect(x - offset, y + offset, offset, offset/2);
rect(x + offset, y + offset, offset, offset/2);
}
```

`drawCar` is the *junction name*, `x` is a *parameter* to the function, and `void` is the *return type*. All the functions I have defined so far did not have a return type; this is precisely what `void` means: no return type. But what is a return type and when might you need one?

Let's recall for a moment the `random()` function examined in Chapter 4. I asked the function for a random number between 0 and some value, and `random()` graciously heeded my request and gave back a random value within the appropriate range. The `random()` function *returned* a value. What type of a value? A floating point number. In the case of `random()`, therefore, its *return type* is a *float*.

The *return type* is the data type that the function returns. In the case of `random()`, I did not specify the return type, however, the creators of Processing did, and it's documented on the reference page for `random()`.

> *Each time the `random()` function is called, it returns an unexpected value within the specified range. If one argument is passed to the function it will return a float between zero and the value of the argument. The function call `random(5)` returns values between 0 and 5. If two arguments are passed, it will return a float with a value between the arguments. The function call `random(-5, 10.2)` returns values between –5 and 10.2.*
>
> *—From http://www.processing.org/reference/random.html*

If you want to write your own function that returns a value, you have to specify the type in the function definition. Let's create a trivially simple example:

```
int sum(int a, int b, int c) {

  int total = a + b + c;
  return total;
}
```

This function, which adds three numbers together, has a return type: `int`.

A return statement is required! A function with a return type must always return a value of that type.

Instead of writing `void` as the return type as I have in previous examples, I now write `int`. This specifies that the function must return a value of type integer. In order for a function to return a value, a *return statement* is required. A return statement looks like this:

`return valueToReturn;`

If you do not include a return statement, Processing will give you an error:

• *This function must return a result of type int.*

As soon as the return statement is executed, the program exits the function and sends the returned value back to the location in the code where the function was called. That value can be used in an assignment operation (to give another variable a value) or in any appropriate expression. See the illustration in Figure 7-5. Here are some examples:

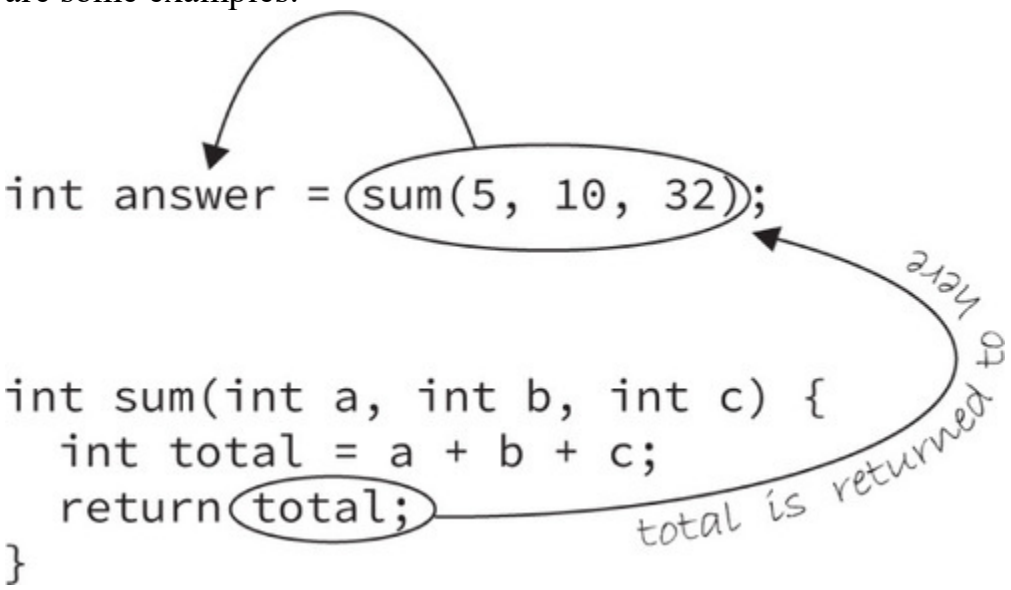


FIGURE 7-5

```
int x = sum(5, 6, 8);
int y = sum(8, 9, 10) * 2;
int z = sum(x, y, 40);
line(100, 100, 110, sum(x, y, z));
```

I hate to bring up playing catch in the park again, but you can think of it as follows. You (*the main program*) throw a copy of a ball to your friend (*a function*). After

your friend catches that ball, he or she thinks for a moment, puts a number inside the ball (*the return value*) and passes it back to you.
Functions that return values are traditionally used to perform complex calculations that may need to be performed multiple times throughout the course of the program. One example is calculating the distance between two points: (x1,y1) and (x2,y2). The distance between pixels is a very useful piece of information in interactive applications. Processing, in fact, has a built-in distance function that you can use. It's called `dist()`.

```
float d = dist(100, 100, mouseX, mouseY);
```

Calculating the distance between `(100,100)` and `(mouseX,mouseY)`.

This line of code calculates the distance between the mouse location and the point (100, 100). For the moment, let's pretend Processing did not include this function in its library. Without it, you would have to calculate the distance manually, using the Pythagorean Theorem, as shown in Figure 7-6.

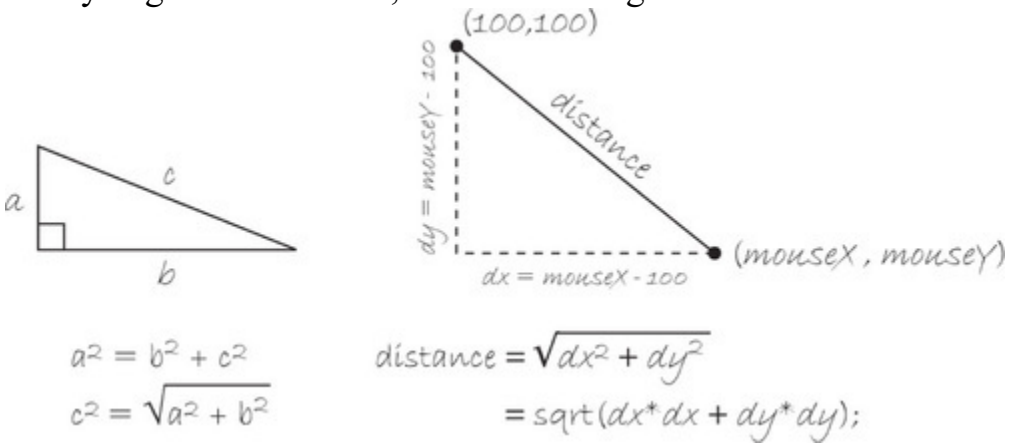


FIGURE 7-6

```
float dx = mouseX - 100;
float dy = mouseY - 100;
float d = sqrt(dx*dx + dy*dy);
```

If you wanted to perform this calculation many times over the course of a program with many different pairs of coordinates, it would be easier to move it into a function that returns the value *d*.

```
float distance(float x1, float y1, float x2, float y2) {
  float dx = x1 - x2;
  float dy = y1 - y2;
  float d = sqrt(dx*dx + dy*dy);
  return d;
}
```

Our version of Processing's `dist()` function.

Note the use of the return type `float`. Again, I do not have to write this function because Processing supplies it for me. But since I did, I can now show an example that makes use of this function.

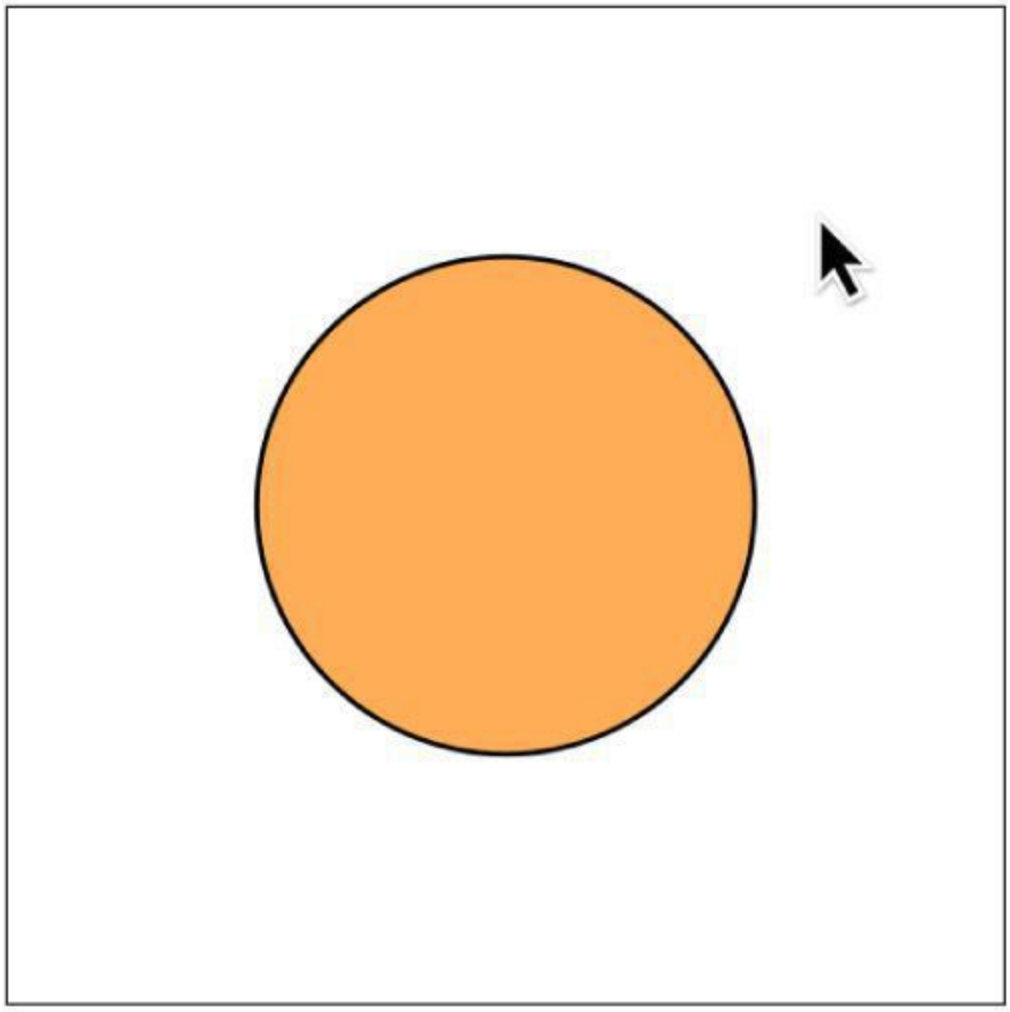


FIGURE 7-7

Example 7-5

Using a function that returns a value, distance

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


void draw() {
  background(255);
  stroke(0);

  float d = distance(width/2, height/2, mouseX, mouseY);

  fill(d*3, d*2, d);
  ellipseMode(CENTER);
  ellipse(width/2, height/2, 100, 100);
}

float distance(float x1, float y1, float x2, float y2) {
  float dx = x1 - x2;
  float dy = y1 - y2;
  float d = sqrt(dx*dx + dy*dy);
  return d;
}
```

The result of the `distance()` function is used to color a circle. I could have used the built-in function `dist()` instead, but I am demonstrating how how to define a function that returns a value.

Exercise 7-8

Write a function that takes one argument — F for Fahrenheit — and computes the result of the following equation (converting the temperature to Celsius). Hint: in Processing if you divide an integer by an integer you will get an integer, same with floating point! In other words, 1/2 evaluates to 0 while 1.0/2.0 evaluates to 0.5.

```
// Formula: C = (F - 32) * (5/9)
______ convertToCelsius(float ________) {
______ _______ = ____________________
_____________________________________
```

```
}
```

7-8 Zoog reorganization

Zoog is now ready for a fairly major overhaul.

• Reorganize Zoog with two functions: `drawZoog()` and `jiggleZoog()`. Just for variety, I am going to have Zoog jiggle (move randomly in both the x and y directions) instead of bouncing back and forth.

• Incorporate parameters so that Zoog's jiggliness is determined by the `mouseX` position and Zoog's eye color is determined by Zoog's distance to the mouse.

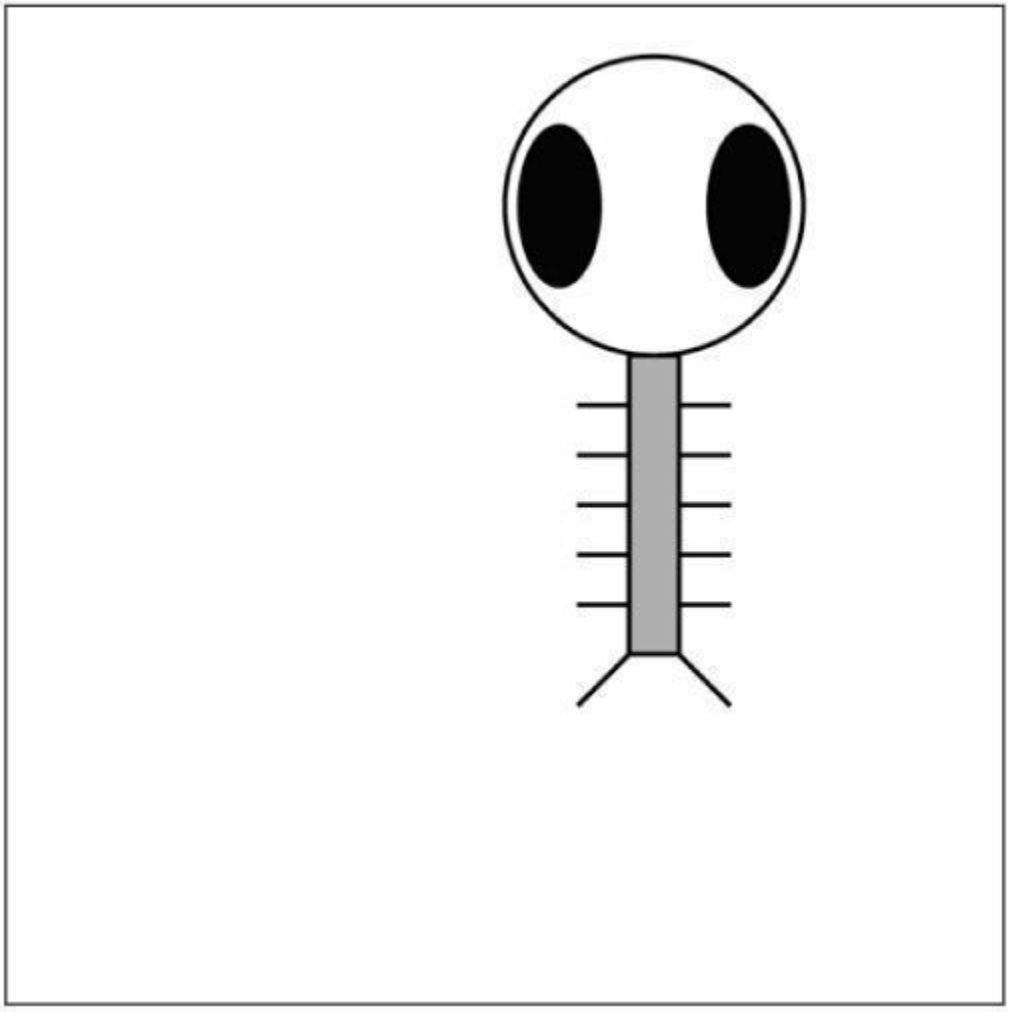


FIGURE 7-8

Example 7-6

Zoog with functions

```
float x = 100;
float y = 100;
float w = 60;
float h = 60;
float eyeSize = 16;

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

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

  // A color based on distance from the mouse
  float d = dist(x, y, mouseX, mouseY);
  color c = color(d);

  // mouseX position determines speed factor for moveZoog function
  float factor = constrain(mouseX/10, 0, 5);

  jiggleZoog(factor);
  drawZoog(c);
```

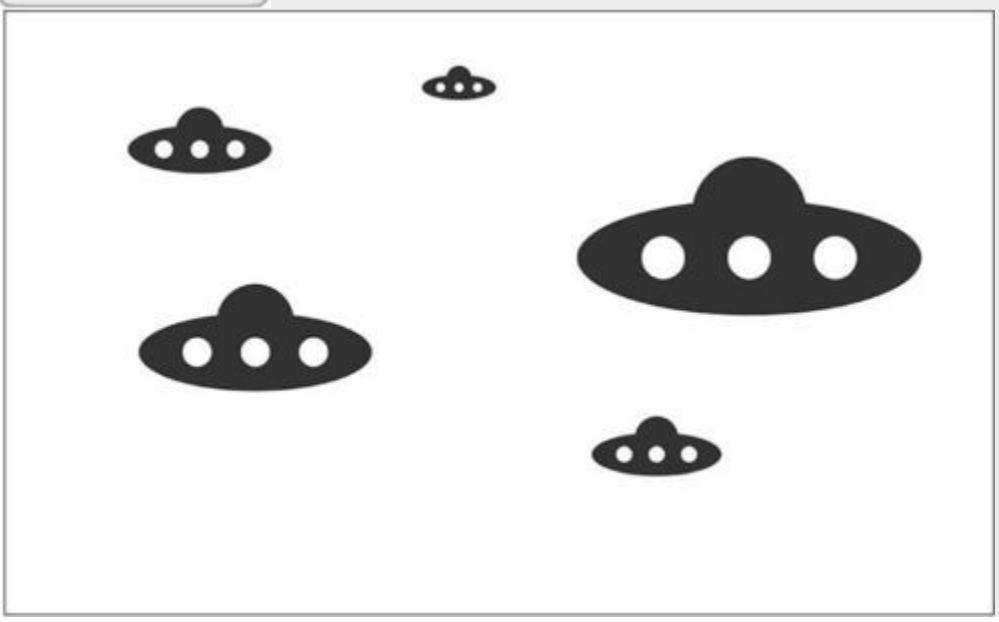
The code for changing the variables associated with Zoog and displaying Zoog is moved outside of `draw()` and into functions called here. The functions are given arguments, such as "jiggle Zoog by the following factor" and "draw Zoog with the following eye color."

```
}

void jiggleZoog(float speed) {
  // Change the x and y location of Zoog randomly
  x = x + random(-1, 1) * speed;
  y = y + random(-1, 1) * speed;

  // Constrain Zoog to window
  x = constrain(x, 0, width);
  y = constrain(y, 0, height);
}

void drawZoog(color eyeColor) {
  // Set ellipses and rects to CENTER mode
  ellipseMode(CENTER);
  rectMode(CENTER);

  // Draw Zoog's arms with a for loop
  for (float i = y - h/3; i < y + h/2; i += 10) {
    stroke(0);
    line(x - w/4, i, x + w/4, i);
  }

  // Draw Zoog's body
  stroke(0);
  fill(175);
  rect(x, y, w/6, h);

  // Draw Zoog's head
  stroke(0);
  fill(255);
  ellipse(x, y - h, w, h);

  // Draw Zoog's eyes
  fill(eyeColor);
  ellipse(x - w/3, y - h, eyeSize, eyeSize*2);
  ellipse(x + w/3, y - h, eyeSize, eyeSize*2);

  // Draw Zoog's legs
  stroke(0);
  line(x - w/12, y + h/2, x - w/4, y + h/2 + 10);
  line(x + w/12, y + h/2, x + w/4, y + h/2 + 10);
}
```

Exercise 7-9: Write a function that draws Zoog based on a set of parameters. Some ideas are: Zoog's x and y coordinates, its width and height, eye color.

Exercise 7-10

Another idea (if you're feeling tired of Zoog) is to create a design for a spaceship and draw several to the screen with slight variations based on arguments you pass to a function. Here's a screenshot and the beginnings of some example code.



```
void setup() {
  size(640, 360);
}

void draw() {
  background(255);
  spaceShip(482, 159, 223);
  spaceShip(126, 89, 93);
  spaceShip(422, 286, 84);
  spaceShip(294, 49, 48);
  spaceShip(162, 220, 151);
}
```

Note the use of only three arguments: x, y, and size. However, you might consider adding parameters for color, number of windows, and more.

_____
_____
_____
_____
_____
_____
_____

Exercise 7-11: Rewrite your Lesson Two Project using functions.