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.
– Parameter passing.
– Returning a value.
– Reusability.

7.1 Break It Down

The examples provided in Chapters 1 through 6 are short. We probably have not looked at 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 we can make interesting visual sketches with small amounts of code. But as we move forward to looking at more complex projects, such as network applications or image processing programs, we will start to have hundreds of lines of code. We will be writing essays, not paragraphs. And these large amounts of code can prove to be unwieldy inside of our two main blocks—***setup()*** and ***draw().***

Functions are a means of taking the parts of our program and separating them out into modular pieces, making our code easier to read, as well as to revise. Let's consider the video game Space Invaders. Our 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 we get into objects in Chapter 8, we will use the term "method" to describe functions inside of objects.

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

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

We are calling functions we made up inside of ***draw()!***

The above demonstrates how functions will make our lives easier with clear and easy to manage code. Nevertheless, we are missing an important piece: the function *definitions*. Calling a function is old hat. We do this all the time when we write ***line()***, ***rect()***, ***fill()***, and so on. Defining a new "made-up" function is going to be hard work.

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

- **Modularity**—Functions break down a larger program into smaller parts, making code more manageable and readable. Once we have figured out how to draw a spaceship, for example, we 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 us to reuse code without having to retype it. What if we want to make a two player Space Invaders game with two spaceships? We can *reuse* the ***drawSpaceShip()*** function by calling it multiple times without having to repeat code over and over.

In this chapter, we will look at some of our previous programs, written without functions, and demonstrate the power of modularity and reusability by incorporating functions. In addition, we 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, we 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 to program the game Pong?*

7.2 "User Defined" Functions

In *Processing*, we have been using functions all along. When we say "line(0,0,200,200);" we 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 we have started to explore throughout the first six chapters of this book. Now it is time to move beyond the built-in functions of *Processing* and write our own *user-defined (AKA "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 (arguments ) {
   // Code body of function
}
```

Deja vu?

Remember when in Chapter 3 we introduced the functions ***setup()*** and ***draw()***? Notice that they follow the same format we are learning now.

setup() and ***draw()*** are functions we define and are called automatically by *Processing* in order to run the sketch. All other functions we write have to be called by us.

For now, let's focus solely on the functionName and code body, ignoring "returnType" and "arguments".

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 (we made it up) and its code body consists of two instructions (we can have as much or as little code as we choose). It is 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 5-6: Bouncing ball

```
// Declare global variables
int x = 0;
int speed = 1;

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

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

```
  // Change x by speed
  x = x + speed;
```

Move the ball!

```
  // If we've reached an edge, reverse speed
  if ((x > width) || (x < 0)) {
  speed = speed * -1;
  }
```

Bounce the ball!

```
  // Display circle at x location
  stroke(0);
  fill(175);
  ellipse(x,100,32,32);
```

Display the ball!

```
}
```

Once we have determined how we want to divide the code up into functions, we 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 7-3: 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);
  smooth();
}

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

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

```
// 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 we've 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);
}
```

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 our bouncing ball function was not behaving appropriately. In order to find the problem, we now have the option of turning on and off parts of the program. For example, we 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, we can more easily deduce 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 we said "Let's ignore **ReturnType** and **Arguments**." We 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 is the concept of *arguments* (AKA "parameters").

Arguments are values that are "passed" into a function. You can think of them as conditions under which the function should operate. Instead of merely saying "Move," a function might say "Move *N* number of steps," where "*N*" is the argument.

When we display an ellipse in *Processing*, we are required to specify details about that ellipse. We can't just say draw an ellipse, we have to say draw an ellipse *at this location* and *with this size*. These are the ***ellipse()*** function's *arguments* and we encountered this in Chapter 1 when we learned to call functions for the first time.

Let's rewrite ***drawBlackCircle()*** to include an argument:

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

"diameter" is an arguments to the function ***drawBlackCircle()***.

An argument is simply a variable declaration inside the parentheses in the function definition. This variable is a *local variable* (Remember our discussion in Chapter 6?) to be used in that function (and only in that function). The white circle will be sized according to the value placed in parentheses.

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

Looking at the bouncing ball example, we could rewrite the ***move()*** function to include an argument:

```
void move(int speedFactor) {
  x = x + (speed * speedFactor);
}
```

The argument "speedFactor" affects how fast the circle moves.

In order to move the ball twice as fast:

```
move(2);
```

Or by a factor of 5:

```
move(5);
```

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

```
move(mouseX/10);
```

Arguments pave the way for more flexible, and therefore reusable, functions. To demonstrate this, we will look at code for drawing a collection of shapes and examine how functions allow us 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):

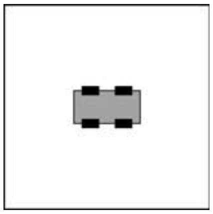


fig. 7.1

```
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, we repeat the above code with different values, as shown in Figure 7.2.

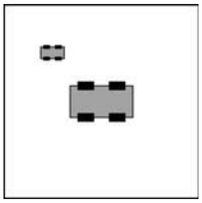


fig. 7.2

```
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, we are doing the same thing twice, why bother repeating all that code? To escape this repetition, we can move the code into a function that displays the car according to several arguments (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);
  rect(x,y,thesize,thesize/2);
  // 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.

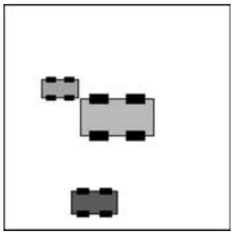


fig. 7.3

In the ***draw()*** function, we then call the ***drawCar()*** function three times, passing four *parameters* each time. See the output in Figure 7.3.

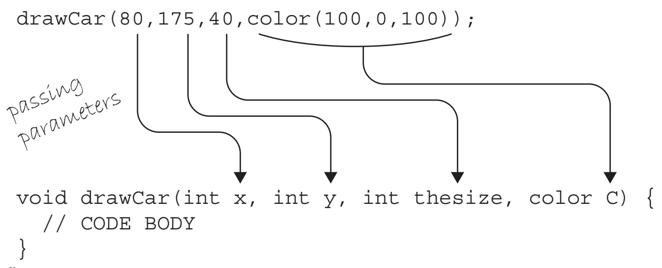
```
void setup() {
  size(200,200);
}

void draw() {
  background(0);
  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, *arguments* are the variables that live inside the parentheses in the function definition, that is, "*void drawCar(int x, int y, int thesize, color c).*" *Parameters* 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 rather trivial and we should not be terribly concerned if we confuse the use of the two words from time to time.

The concept to focus on is this ability to *pass* parameters. We will not be able to advance our programming knowledge unless we are comfortable with this technique.

Let's go with the word *pass*. Imagine a lovely, sunny day and you are 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


drawCar(80,175,40,color(100,0,100));
passing parameters
void drawCar(int x, int y, int thesize, color C) {
// CODE BODY
}
fig. 7.4

(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.

Important Things to Remember about Passing Parameters

- You must pass the same number of parameters as defined in the function.
- When a parameter is passed, it must be of the same *type* as declared within the arguments 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 as a parameter 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.)
- Arguments 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: Here is the bouncing ball from Example 5-6 combined with the **drawCar()** function. Fill in the blanks so that you now have a bouncing car with parameter passing! (Note that the global variables are now named globalX and globalY to avoid confusion with the local variables x and y in **drawCar()**).*

```
int globalX = 0;
int globalY = 100;
int speed = 1;

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

void draw() {
  background(0);
  ____________________________________________________

  ____________________________________________________

  ____________________________________________________
}

void move() {
  // Change the x location by speed
  globalX = globalX + speed;
}

void bounce() {
  if ((globalX > width) || (globalX < 0)) {
    speed = speed * -1;
  }
}

void drawCar(int x, int y, int thesize, color c) {
  int offset = thesize / 4;
  rectMode(CENTER);
  stroke(200);
  fill(c);
  rect(x,y,thesize,thesize/2);
  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);
    }
```

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 (integer, 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 is not so trivial when passing a variable.

The following code has a function entitled ***randomizer()*** that receives one argument (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 argument **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 is more commonly referred to as "pass by value." This holds true for all primitive data types (the only kinds we know about so far: integer, float, etc.), but will not be the case when we learn about *objects* in the next chapter.

This example also gives us 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 above 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");
}
```

Output:

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

It is perfectly reasonable to call a function from within a function. In fact, we do this all the time whenever we call any function from inside of ***setup()*** or ***draw()***.

7.7 Return Type

So far we have seen how functions can separate a sketch into smaller parts, as well as incorporate arguments to make it reusable. There is one piece missing still, however, 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 (Arguments) {
//code body of function
}

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

```
// A function to move the ball
void move(int speedFactor) {
  // Change the x location of organism by speed multiplied by speedFactor
  x = x + (speed * speedFactor);
}
```

"move" is the **FunctionName**, "speedFactor" is an **Argument** to the function and "void" is the **ReturnType**. All the functions we 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 we need one?

Let's recall for a moment the ***random()*** function we examined in Chapter 4. We asked the function for a random number between 0 and some value, and ***random()*** graciously heeded our request and gave us 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()***, we did not specify the return type, however, the creators of *Processing* did, and it is 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 parameter is passed to the function it will return a float between zero and the value of the parameter. The function call **random(5)** returns values between 0 and 5. If two parameters are passed, it will return a float with a value between the parameters. The function call **random(–5, 10.2)** returns values between –5 and 10.2.*

—From http://www.processing.org/reference/random.html

If we want to write our own function that returns a value, we 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 we have in previous examples, we 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:

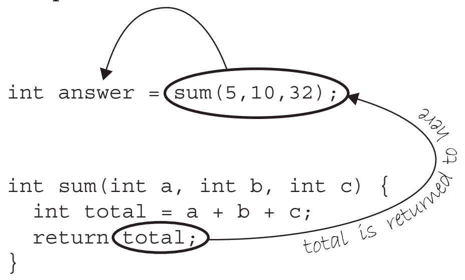
return valueToReturn;

If we did not include a return statement, *Processing* would give us an error:

> ***The method "int sum(int a, int b, int c);" must contain a return statement with an expression compatible with 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:

```
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));
```



```
int answer = sum(5,10,32);


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

fig. 7.5

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 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 we can use. It is 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, we would have to calculate the distance manually, using the Pythagorean Theorem, as shown in Figure 7.6.

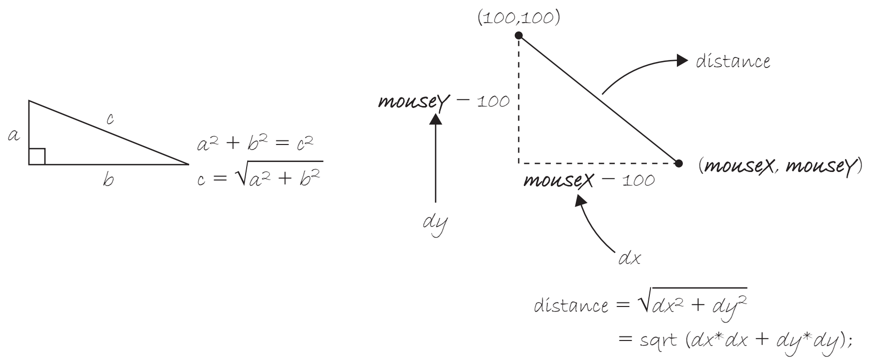


fig. 7.6

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

If we wanted to perform this calculation many times over the course of a program, 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, we do not have to write this function because *Processing* supplies it for us. But since we did, we can now look at an example that makes use of this function.

Example 7-4: Using a function that returns a value, distance

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

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

// Top left square
fill(distance(0,0,mouseX,mouseY));
rect(0,0,width/2-1,height/2-1);

// Top right square
fill(distance(width,0,mouseX,mouseY));
rect(width/2,0,width/2-1,height/2-1);

// Bottom left square
fill(distance(0,height,mouseX,mouseY));
rect(0,height/2,width/2-1,height/2-1);

// Bottom right square
fill(distance(width,height,mouseX,mouseY));
rect(width/2,height/2,width/2-1,height/2-1);
}

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 distance function is used to calculate a brightness value for each quadrant. We could use the built-in function ***dist()*** instead, but we are learning how to write our own functions.

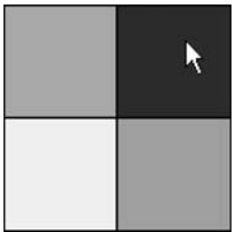


fig. 7.7

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).

C = (F − 32) * (5/9)

```
______ tempConverter(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, we are going to have Zoog jiggle (move randomly in both the *x* and *y* directions) instead of bouncing back and forth.
- Incorporate arguments 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.

Example 7-5: Zoog with functions

```
float x = 100;
float y = 100;
float w = 60;
float h = 60;
float eyeSize = 16;
void setup() {
  size(200,200);
  smooth();
}

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

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

  // pass in a color to drawZoog
  // function for eye's color
  float d = dist(x,y,mouseX,mouseY);
  color c = color(d);
  drawZoog(c);
}
```

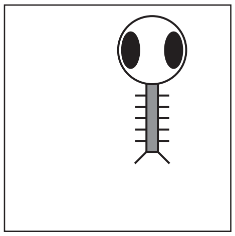


fig. 7.8

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: Following is a version of Example 6-11 ("multiple Zoogs") that calls a function to draw Zoog. Write the function definition that completes this sketch. Feel free to redesign Zoog in the process.

```
void setup() {
  size(400,200);  // Set the size of the window
  smooth();       // Enables Anti-Aliasing (smooth edges on
                     shapes)
}
```

```
void draw() {
  background(0);  // Draw a black background
  int y = height/2;
  // Multiple versions of Zoog are displayed by using a for loop
  for (int x = 80; x < width; x += 80) {
    drawZoog(x,100,60,60,16);
  }
}
```

Exercise 7-10: Rewrite your Lesson Two Project using functions. Still 10! A copy was sent into newnum so num has not changed.

This page intentionally left blank