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;

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

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

}
```

Move the ball!

Bounce the ball!

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):

```
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);
```

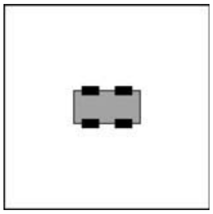


fig. 7.1

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.

```
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);
```

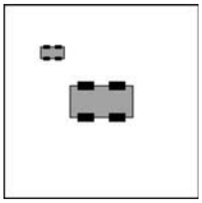


fig. 7.2

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.

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

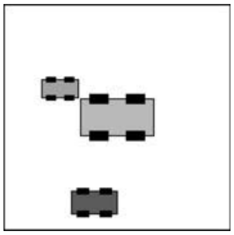


fig. 7.3

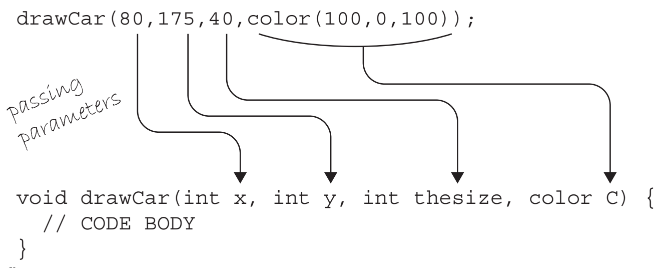
```
void setup() {
  size(200,200);
}

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));

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:

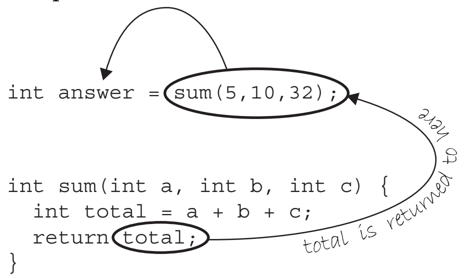
return valueToReturn;

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: (*x*1,*y*1) and (*x*2,*y*2). 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.

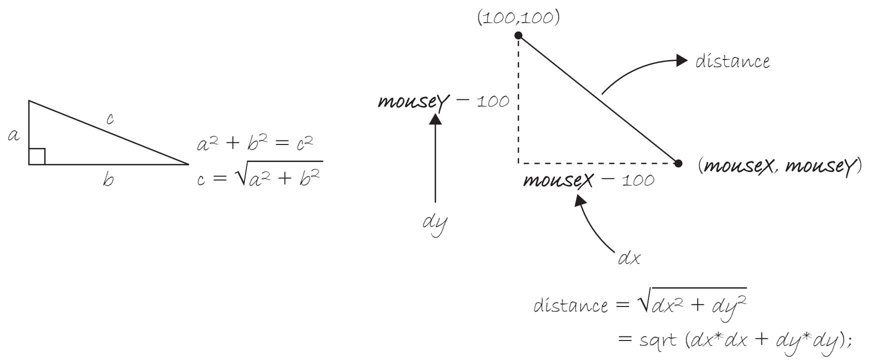


fig. 7.6

```
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.

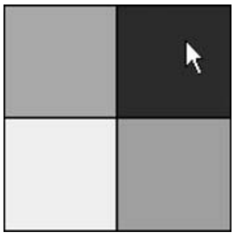


fig. 7.7

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);
}
```

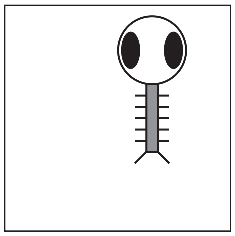


fig. 7.8

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

8 Objects

"No object is so beautiful that, under certain conditions, it will not look ugly."
—Oscar Wilde

In this chapter:
– Data and functionality, together at last.
– What is an object?
– What is a class?
– Writing your own classes.
– Creating your own objects.
– Processing "tabs."

8.1 I'm down with OOP.

Before we begin examining the details of how object-oriented programming (OOP) works in *Processing*, let's embark on a short conceptual discussion of "objects" themselves. It is important to understand that we are not introducing any new programming fundamentals: objects use everything we have already learned: variables, conditional statements, loops, functions, and so on. What is entirely new, however, is a way of thinking, a way of structuring and organizing everything we have already learned.

Imagine you were not programming in *Processing*, but were instead writing out a program for your day, a list of instructions, if you will. It might start out something like:

- Wake up.
- Drink coffee (or tea).
- Eat breakfast: cereal, blueberries, and soy milk.
- Ride the subway.

What is involved here? Specifically, what *things* are involved? First, although it may not be immediately apparent from how we wrote the above instructions, the main thing is *you*, a human being, a person. You exhibit certain properties. You look a certain way; perhaps you have brown hair, wear glasses, and appear slightly nerdy. You also have the ability to do stuff, such as wake up (presumably you can also sleep), eat, or ride the subway. An object is just like you, a thing that has properties and can do stuff.

So how does this relate to programming? The properties of an object are variables; and the stuff an object can do are functions. Object-oriented programming is the marriage of everything we have learned in Chapters 1 through 7, data and functionality, all rolled into one *thing*.

Let's map out the data and functions for a very simple human object:

Human data

- Height.
- Weight.
- Gender.

- Eye color.
- Hair color.

Human functions

- Sleep.
- Wake up.
- Eat.
- Ride some form of transportation.

Now, before we get too much further, we need to embark on a brief metaphysical digression. The above structure is not a human being itself; it simply describes the idea, or the concept, behind a human being. It describes what it is to be human. To be human is to have height, hair, to sleep, to eat, and so on. This is a crucial distinction for programming objects. This human being template is known as a *class*. A *class* is different from an *object*. You are an object. I am an object. That guy on the subway is an object. Albert Einstein is an object. We are all people, real world *instances* of the idea of a human being.

Think of a cookie cutter. A cookie cutter makes cookies, but it is not a cookie itself. The cookie cutter is the *class*, the cookies are the *objects*.

Exercise 8-1: Consider a car as an object. What data would a car have? What functions would it have?

Car data	*Car functions*

8.2 Using an Object

Before we look at the actual writing of a *class* itself, let's briefly look at how using objects in our main program (i.e., ***setup()*** and ***draw()***) makes the world a better place.

Returning to the car example from Chapter 7, you may recall that the pseudocode for the sketch looked something like this:

***Data (Global Variables)*:**

```
Car color.
Car x location.
Car y location.
Car x speed.
```

***Setup*:**

```
Initialize car color.
Initialize car location to starting point.
Initialize car speed.
```

***Draw*:**

```
Fill background.
Display car at location with color.
Increment car's location by speed.
```

In Chapter 7, we defined global variables at the top of the program, initialized them in ***setup()***, and called *functions* to move and display the car in ***draw()***.

Object-oriented programming allows us to take all of the variables and functions out of the main program and store them inside a car object. A car object will know about its data—*color*, *location*, *speed*. That is part one. Part two of the car object is the stuff it can do, the methods (functions inside an object). The car can *move* and it can be *displayed*.

Using object-oriented design, the pseudocode improves to look something like this:

***Data (Global Variables)*:**

```
Car object.
```

***Setup*:**

```
Initialize car object.
```

***Draw*:**

```
Fill background.
Display car object.
Move car object.
```

Notice we removed all of the global variables from the first example. Instead of having separate variables for car color, car location, and car speed, we now have only one variable, a ***Car*** variable! And instead of initializing those three variables, we initialize one thing, the ***Car*** object. Where did those variables go? They still exist, only now they live inside of the ***Car*** object (and will be defined in the ***Car*** class, which we will get to in a moment).

Moving beyond pseudocode, the actual body of the sketch might look like:

```
Car myCar;

void setup() {
  myCar = new Car();
}

void draw() {
  background(0);
  myCar.move();
  myCar.display();
}
```

An object in *Processing*.

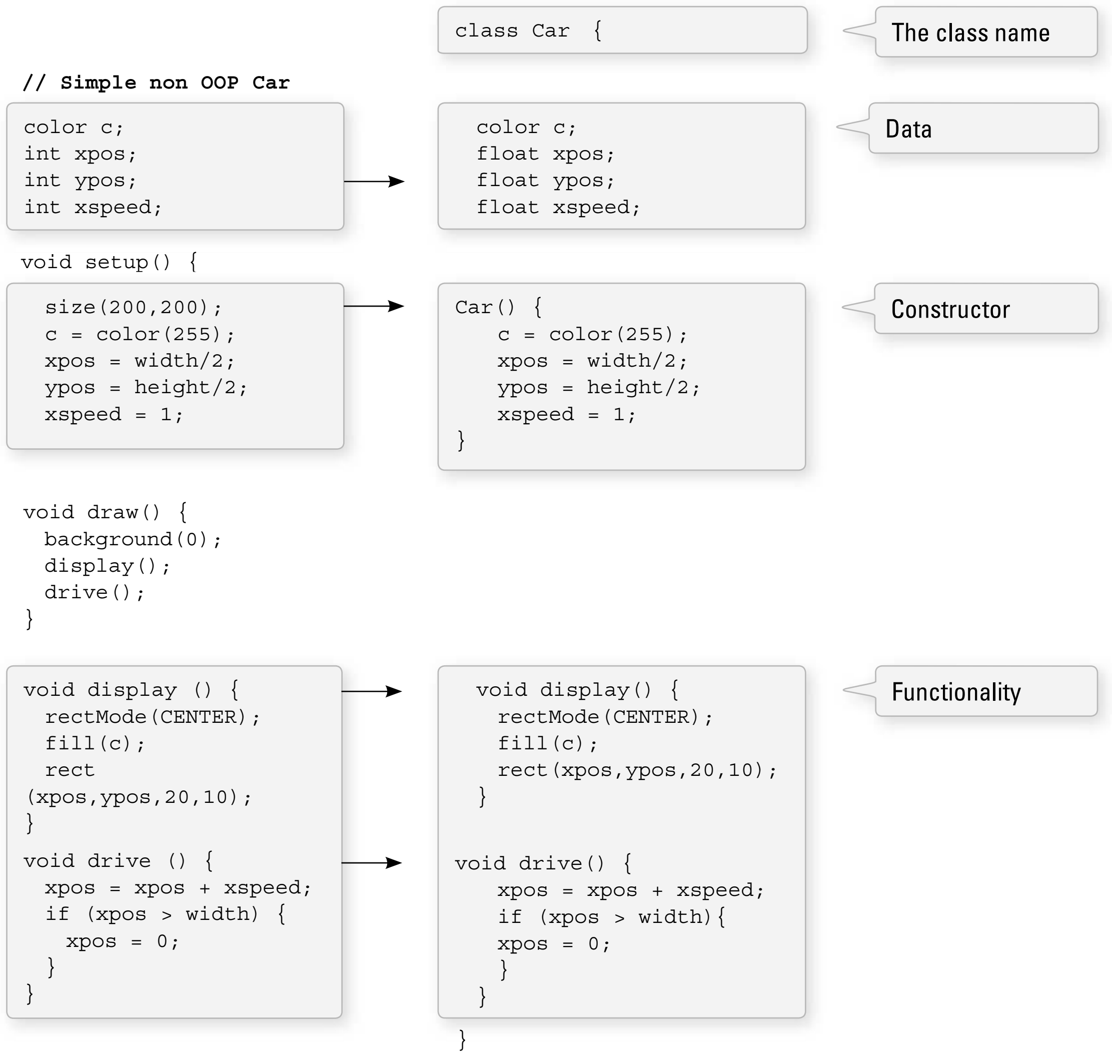
We are going to get into the details regarding the previous code in a moment, but before we do so, let's take a look at how the Car *class* itself is written.

8.3 Writing the Cookie Cutter

The simple Car example above demonstrates how the use of object in *Processing* makes for clean, readable code. The hard work goes into writing the object template, that is the *class* itself. When you are first learning about object-oriented programming, it is often a useful exercise to take a program written without objects and, not changing the functionality at all, rewrite it using objects. We will do exactly this with the car example from Chapter 7, recreating exactly the same look and behavior in an object-oriented manner. And at the end of the chapter, we will remake Zoog as an object.

All classes must include four elements: *name*, *data*, *constructor*, and *methods*. (Technically, the only actual required element is the class name, but the point of doing object-oriented programming is to include all of these.)

Here is how we can take the elements from a simple non-object-oriented sketch (a simplified version of the solution to Exercise 7-6) and place them into a Car class, from which we will then be able to make Car objects.



```
// Simple non OOP Car

color c;
int xpos;
int ypos;
int xspeed;

void setup() {

  size(200,200);
  c = color(255);
  xpos = width/2;
  ypos = height/2;
  xspeed = 1;

void draw() {
  background(0);
  display();
  drive();
}

void display () {
  rectMode(CENTER);
  fill(c);
  rect
(xpos,ypos,20,10);
}

void drive () {
  xpos = xpos + xspeed;
  if (xpos > width) {
    xpos = 0;
  }
}
```

```
class Car {

  color c;
  float xpos;
  float ypos;
  float xspeed;

Car() {
    c = color(255);
    xpos = width/2;
    ypos = height/2;
    xspeed = 1;
}

  void display() {
    rectMode(CENTER);
    fill(c);
    rect(xpos,ypos,20,10);
  }

void drive() {
    xpos = xpos + xspeed;
    if (xpos > width){
    xpos = 0;
    }
  }
}
```

- **The Class Name**—The name is specified by "class WhateverNameYouChoose". We then enclose all of the code for the class inside curly brackets after the name declaration. Class names are traditionally capitalized (to distinguish them from variable names, which traditionally are lowercase).
- **Data**—The data for a class is a collection of variables. These variables are often referred to as *instance* variables since each *instance* of an object contains this set of variables.
- **A Constructor**—The constructor is a special function inside of a class that creates the instance of the object itself. It is where you give the instructions on how to set up the object. It is just like *Processing*'s ***setup()*** function, only here it is used to create an individual object within the sketch, whenever a ***new*** object is created from this *class*. It always has the same name as the class and is called by invoking the ***new*** operator: "***Car myCar = new Car();***".
- **Functionality**—We can add functionality to our object by writing methods. These are done in the same way as described in Chapter 7, with a return type, name, arguments, and a body of code.

This code for a *class* exists as its own block and can be placed anywhere outside of ***setup()*** and ***draw()***.

A Class Is a New Block of Code!

```
void setup() {

}
void draw() {

}
class Car {

}
```

Exercise 8-2: Fill in the blanks in the following Human class definition. Include a function called ***sleep()*** *or make up your own function. Follow the syntax of the Car example. (There are no right or wrong answers in terms of the actual code itself; it is the structure that is important.)*

```
_________ _________ {
   color hairColor;
   float height;
  _________() {
    ____________________________
    ____________________________
}

    ____________________________ {
    ____________________________
    ____________________________
  }
}
```

8.4 Using an Object: The Details

In Section 8.2, we took a quick peek at how an object can greatly simplify the main parts of a *Processing* sketch (***setup()*** and ***draw()***).

```
Car myCar;

void setup() {
  myCar = new Car();
}

void draw() {
  background(0);
  myCar.move();
  myCar.display();
}
```

Step 1. Declare an object.

Step 2. Initialize object.

Step 3. Call methods on the object.

Let's look at the details behind the above three steps outlining how to use an object in your sketch.

Step 1. Declaring an object variable.

If you flip back to Chapter 4, you may recall that a variable is declared by specifying a *type* and a *name*.

```
// Variable Declaration
int var;   // type name
```

The above is an example of a variable that holds onto *a primitive*, in this case an integer. As we learned in Chapter 4, primitive data types are singular pieces of information: an integer, a float, a character. Declaring a variable that holds onto an object is quite similar. The difference is that here the type is the class name, something we will make up, in this case "Car." Objects, incidentally, are not primitives and are considered *complex* data types. (This is because they store multiple pieces of information: data and functionality. Primitives only store data.)

Step 2. Initializing an object.

Again, you may recall from Chapter 4 that in order to initialize a variable (i.e., give it a starting value), we use an assignment operation—variable equals something.

```
// Variable Initialization
var = 10;   // var equals 10
```

Initializing an object is a bit more complex. Instead of simply assigning it a primitive value, like an integer or floating point number, we have to construct the object. An object is made with the ***new*** operator.

```
// Object Initialization
myCar = new Car();
```

The ***new*** operator is used to make a new object.

In the above example, "myCar" is the object variable name and "=" indicates we are setting it equal to something, that something being a ***new*** instance of a Car object. What we are really doing here is initializing a Car object. When you initialize a primitive variable, such as an integer, you just set it equal to a number. But an object may contain multiple pieces of data. Recalling the Car class from the previous section, we see that this line of code calls the *constructor*, a special function named ***Car()*** that initializes all of the object's variables and makes sure the Car object is ready to go.

One other thing; with the primitive integer "var," if you had forgotten to initialize it (set it equal to 10), *Processing* would have assigned it a default value, zero. An object (such as "myCar"), however, has no default value. If you forget to initialize an object, *Processing* will give it the value *null. null* means *nothing*. Not zero. Not negative one. Utter nothingness. Emptiness. If you encounter an error in the message window that says "***NullPointerException***" (and this is a pretty common error), that error is most likely caused by having forgotten to initialize an object. (See the Appendix for more details.)

Step 3. Using an object

Once we have successfully declared and initialized an object variable, we can use it. Using an object involves calling functions that are built into that object. A human object can eat, a car can drive, a dog can bark. Functions that are inside of an object are technically referred to as "methods" in Java so we can begin to use this nomenclature (see Section 7.1). Calling a method inside of an object is accomplished via dot syntax:

variableName.objectMethod(Method Arguments);

In the case of the car, none of the available functions has an argument so it looks like:

```
myCar.draw();
myCar.display();
```

Functions are called with the "dot syntax".

Exercise 8-3: Assume the existence of a Human class. You want to write the code to declare a Human object as well as call the function ***sleep()*** *on that human object. Write out the code below:*

Declare and initialize the Human object: ________________________________

Call the ***sleep()*** function: ________________________________

8.5 Putting It Together with a Tab

Now that we have learned how to define a class and use an object born from that class, we can take the code from Sections 8.2 and 8.3 and put them together in one program.

Example 8-1: A Car class and a Car object

```
Car myCar;

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

  // Initialize Car object
  myCar = new Car();
}

void draw() {
  background(0);
  // Operate Car object.
  myCar.move();
  myCar.display();
}
```

Declare car object as a globle variable.

Initialize car object in ***setup()*** by calling constructor.

Operate the car object in ***draw()*** by calling object methods using the dots syntax.

```
class Car {                  Define a class below the rest of the program.

  color c;
  float xpos;                Variables.
  float ypos;
  float xspeed;

  Car() {                    A constructor.
    c = color(255);
    xpos = width/2;
    ypos = height/2;
    xspeed = 1;
  }

  void display() {           Function.
    // The car is just a square
    rectMode(CENTER);
    fill(c);
    rect(xpos,ypos,20,10);
  }

  void move() {              Function.
    xpos = xpos + xspeed;
    if (xpos > width) {
      xpos = 0;
    }
  }
}
```

You will notice that the code block that contains the Car class is placed below the main body of the program (under ***draw()***). This spot is identical to where we placed user-defined functions in Chapter 7. Technically speaking, the order does not matter, as long as the blocks of code (contained within curly brackets) remain intact. The Car class could go above ***setup()*** or it could even go between ***setup()*** and ***draw()***. Though any placement is technically correct, when programming, it is nice to place things where they make the most logical sense to our human brains, the bottom of the code being a good starting point. Nevertheless, *Processing* offers a useful means for separating blocks of code from each other through the use of tabs.

In your *Processing* window, look for the arrow inside a square in the top right-hand corner. If you click that button, you will see that it offers the "New Tab" option shown in Figure 8.1.

Upon selecting "New Tab," you will be prompted to type in a name for the new tab, as shown in Figure 8.2.

Although you can pick any name you like, it is probably a good idea to name the tab after the *class* you intend to put there. You can then type the main body of code on one tab (entitled "objectExample" in Figure 8.2) and type the code for your class in another (entitled "Car").

Toggling between the tabs is simple, just click on the tab name itself, as shown in Figure 8.3. Also, it should be noted that when a new tab is created, a new .pde file is created inside the sketch folder, as shown in Figure 8.4. The program has both an objectExample.pde file and Car.pde file.

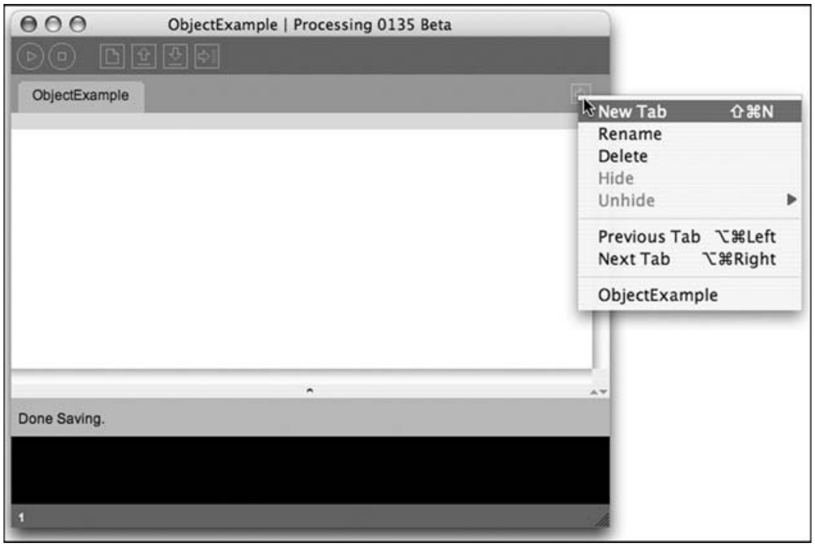
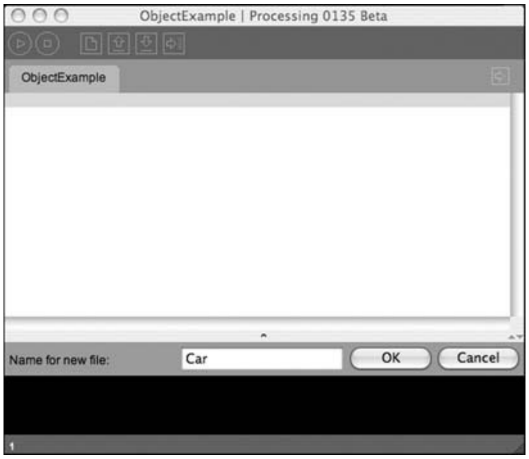
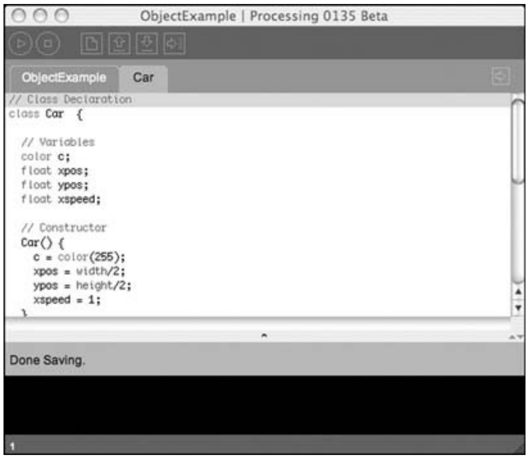


fig. 8.1



Name for new file: Car

fig. 8.2



```
// Class Declaration
class Car  {

  // Variables
  color c;
  float xpos;
  float ypos;
  float xspeed;

  // Constructor
  Car() {
    c = color(255);
    xpos = width/2;
    ypos = height/2;
    xspeed = 1;
  }
```

Done Saving.

fig. 8.3

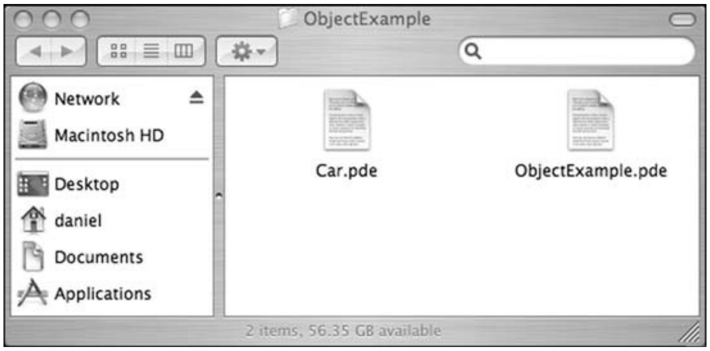


fig. 8.4

Exercise 8-4: Create a sketch with multiple tabs. Try to get the Car example to run without any errors.

8.6 Constructor Arguments

In the previous examples, the car object was initialized using the ***new*** operator followed by the *constructor* for the class.

```
Car myCar = new Car();
```

This was a useful simplification while we learned the basics of OOP. Nonetheless, there is a rather serious problem with the above code. What if we wanted to write a program with two car objects?

```
// Creating two car objects
Car myCar1 = new Car();
Car myCar2 = new Car();
```

This accomplishes our goal; the code will produce two car objects, one stored in the variable myCar1 and one in myCar2. However, if you study the Car class, you will notice that these two cars will be identical: each one will be colored white, start in the middle of the screen, and have a speed of 1. In English, the above reads:

Make a new car.

We want to instead say:

Make a new red car, at location (0,10) with a speed of 1.

So that we could also say:

Make a new blue car, at location (0,100) with a speed of 2.

We can do this by placing arguments inside of the constructor method.

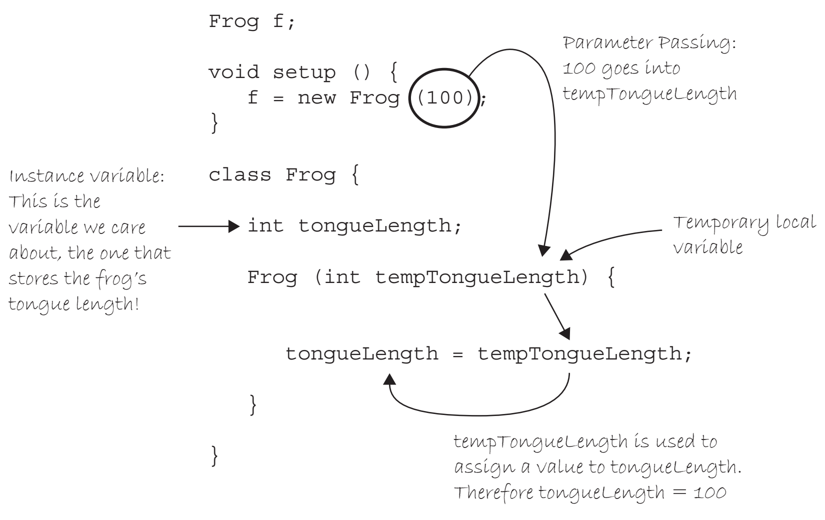
```
Car myCar = new Car(color(255,0,0),0,100,2);
```

The constructor must be rewritten to incorporate these arguments:

```
Car(color tempC, float tempXpos, float tempYpos, float tempXspeed) {
  c = tempC;
  xpos = tempXpos;
  ypos = tempYpos;
  xspeed = tempXspeed;
}
```

In my experience, the use of constructor arguments to initialize object variables can be somewhat bewildering. Please do not blame yourself. The code is strange-looking and can seem awfully redundant: "For every single variable I want to initialize in the constructor, I have to duplicate it with a temporary argument to that constructor?"

Nevertheless, this is quite an important skill to learn, and, ultimately, is one of the things that makes object-oriented programming powerful. But for now, it may feel painful. Let's briefly revisit parameter passing again to understand how it works in this context. See Figure 8.5.



```
Frog f;

void setup () {
   f = new Frog (100);
}

class Frog {

   int tongueLength;

   Frog (int tempTongueLength) {

      tongueLength = tempTongueLength;

   }

}
```

Parameter Passing: 100 goes into tempTongueLength
Instance variable: This is the variable we care about, the one that stores the frog's tongue length!
Temporary local variable
tempTongueLength is used to assign a value to tongueLength. Therefore tongueLength = 100

Translation: Make a new frog with a tongue length of 100.

fig. 8.5

Arguments are local variables used inside the body of a function that get filled with values when the function is called. In the examples, they have *one purpose only*, to initialize the variables inside of an object. These are the variables that count, the car's actual car, the car's actual *x* location, and so on. The constructor's arguments are just *temporary*, and exist solely to pass a value from where the object is made into the object itself.

This allows us to make a variety of objects using the same constructor. You might also just write the word *temp* in your argument names to remind you of what is going on (c vs. tempC). You will also see programmers use an underscore (c vs. c_) in many examples. You can name these whatever you want, of course. However, it is advisable to choose a name that makes sense to you, and also to stay consistent.

We can now take a look at the same program with multiple object instances, each with unique properties.

Example 8-2: Two Car objects

```
Car myCar1;
Car myCar2;

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

  myCar1 = new Car(color(255,0,0),0,100,2);
  myCar2 = new Car(color(0,0,255),0,10,1);
}

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

Two objects!

Parameters go inside the parentheses when the object is constructed.

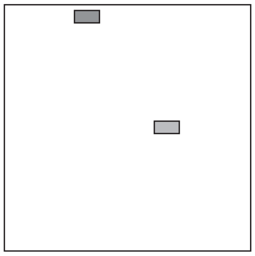


fig. 8.6

```
  myCar1.move();
  myCar1.display();
  myCar2.move();
  myCar2.display();
}

class Car {

  color c;
  float xpos;
  float ypos;
  float xspeed;

  Car(color tempC, float tempXpos, float tempYpos, float tempXspeed) {
    c = tempC;
    xpos = tempXpos;
    ypos = tempYpos;
    xspeed = tempXspeed;
  }

  void display() {
    stroke(0);
    fill(c);
    rectMode(CENTER);
    rect(xpos,ypos,20,10);
  }

  void move() {
    xpos = xpos + xspeed;
    if (xpos > width) {
      xpos = 0;
    }
  }
}
```

Even though there are *multiple* objects, we still only need *one* class. No matter how many cookies we make, only one cookie cutter is needed.Isn't object-oriented programming swell?

The Constructor is defined with arguments.

Exercise 8-5: Rewrite the gravity example from Chapter 5 using objects with a Ball class. Include two instances of a Ball object. The original example is included here for your reference with a framework to help you get started.

```
_______ _______;
Ball ball2;

float grav = 0.1;

void setup() {
  size(200,200);
  ball1 = new _______(50,0,16);
  _________________(100,50,32);
}
```

```
void draw() {
  background(100);
  ball1.display();
  ______________________________
  ______________________________
  ______________________________
}

_________________ {
  float x;
  _____________________
  float speed;
  float w;
  ______(______,______,______) {
    x = ______;
    ____________
    ____________
    speed = 0;
  }
  void ____________() {
    ________________________________
    ________________________________
    ________________________________
  }

  ________________________________
  ________________________________
  ________________________________
  ________________________________
  ________________________________
  ________________________________
}
```

```
// Simple gravity
float x = 100;      // x
location
float y = 0;        // y
location

float speed = 0;    // speed
float gravity = 0.1;// gravity

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

void draw() {
  background(100);

  // display the square
  fill(255);
  noStroke();
  rectMode(CENTER);
  rect(x,y,10,10);

  // Add speed to y location
  y = y + speed;
  // Add gravity to speed
  speed = speed + gravity;

  // If square reaches the bottom
  // Reverse speed
  if (y > height) {
    speed = speed * -0.95;
  }
}
```

8.7 Objects are data types too!

This is our first experience with object-oriented programming, so we want to take it easy. The examples in this chapter all use just one class and make, at most, two or three objects from that class. Nevertheless, there are no actual limitations. A *Processing* sketch can include as many classes as you feel like writing. If you were programming the Space Invaders game, for example, you might create a *Spaceship* class, an *Enemy* class, and a *Bullet* class, using an object for each entity in your game.

In addition, although not *primitive*, classes are data types just like integers and floats. And since classes are made up of data, an object can therefore contain other objects! For example, let's assume you had just finished programming a *Fork* and *Spoon* class. Moving on to a *PlaceSetting* class, you would likely include variables for both a *Fork* object and a *Spoon* object inside that class itself. This is perfectly reasonable and quite common in object-oriented programming.

```
class PlaceSetting {

  Fork fork;
  Spoon spoon;

  PlaceSetting() {
    fork = new Fork();
    spoon = new Spoon();
  }
}
```

A class can include other objects among its variables.

Objects, just like any data type, can also be passed in as arguments to a function. In the Space Invaders game example, if the spaceship shoots the bullet at the enemy, we would probably want to write a function inside the Enemy class to determine if the Enemy had been hit by the bullet.

```
void hit(Bullet b) {
//  Code to determine if
//  the bullet struck the enemy
}
```

A function can have an object as its argument.

In Chapter 7, we showed how when a primitive value (integer, float, etc.) is passed in a function, a copy is made. With objects, this is not the case, and the result is a bit more intuitive. If changes are made to an object after it is passed into a function, those changes will affect that object used anywhere else throughout the sketch. This is known as *pass by reference* since instead of a copy, a reference to the actual object itself is passed into the function.

As we move forward through this book and our examples become more advanced, we will begin to see examples that use multiple objects, pass objects into functions, and more. The next chapter, in fact, focuses on how to make lists of objects. And Chapter 10 walks through the development of a project that includes multiple classes. For now, as we close out the chapter with Zoog, we will stick with just one class.

8.8 Object-Oriented Zoog

Invariably, the question comes up: "When should I use object-oriented programming?" For me, the answer is *always.* Objects allow you to organize the concepts inside of a software application into modular, reusable packages. You will see this again and again throughout the course of this book. However, it is not always convenient or necessary to start out every project using object-orientation, especially while you are learning. *Processing* makes it easy to quickly "sketch" out visual ideas with non object-oriented code.

For any *Processing* project you want to make, my advice is to take a step-by-step approach. You do not need to start out writing classes for everything you want to try to do. Sketch out your idea first by writing code in ***setup()*** and ***draw()***. Nail down the logic of what you want to do as well as how you want it to look. As your project begins to grow, take the time to reorganize your code, perhaps first with functions, then with objects. It is perfectly acceptable to dedicate a significant chunk of your time to this reorganization process (often referred to as *refactoring*) without making any changes to the end result, that is, what your sketch looks like and does on screen.

This is exactly what we have been doing with cosmonaut Zoog from Chapter 1 until now. We sketched out Zoog's look and experimented with some motion behaviors. Now that we have something, we can take the time to *refactor* by making Zoog into an object. This process will give us a leg up in programming Zoog's future life in more complex sketches.

And so it is time to take the plunge and make a Zoog class. Our little Zoog is almost all grown up. The following example is virtually identical to Example 7-5 (Zoog with functions) with one major difference. All of the variables and all of the functions from Example 7-5 are now incorporated into the Zoog class with ***setup()*** and ***draw()*** containing barely any code.

Example 8-3

```
Zoog zoog;

void setup() {
  size(200,200);
  smooth();
  zoog = new Zoog(100,125,60,60,16);
}

void draw() {
  background(255);
  // mouseX position determines speed factor
  float factor = constrain(mouseX/10,0,5);
  zoog.jiggle(factor);
  zoog.display();
}

class Zoog {
  // Zoog's variables
  float x,y,w,h,eyeSize;

  // Zoog constructor
  Zoog(float tempX, float tempY, float tempW, float tempH, float tempEyeSize) {
    x = tempX;
    y = tempY;
    w = tempW;
    h = tempH;
    eyeSize = tempEyeSize;
  }
```

Zoog is an object!

Zoog is given initial properties via the constructor.

Zoog can do stuff with functions!

Everything about Zoog is contained in this one class. Zoog has properties (location, with , height, eye size) and Zoog has abilities (jiggle, display).

```
  // Move Zoog
  void jiggle(float speed) {
    // Change the 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);
  }

  // Display Zoog
  void display() {
    // 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(0);
    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);

  }
}
```

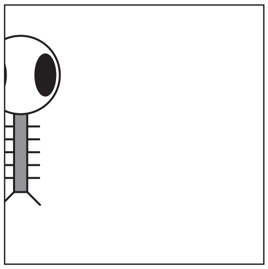


fig. 8.7

Exercise 8-6: Rewrite Example 8-3 to include two Zoogs. Can you vary their appearance? Behavior? Consider adding color as a Zoog variable.

Lesson Three Project

Step 1. Take your Lesson Two Project and reorganize the code using functions.

Step 2. Reorganize the code one step further using a class and an object variable.

Step 3. Add arguments to the Constructor of your class and try making two or three objects with different variables.

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