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;          An object in Processing.

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

void draw() {
  background(0);
  myCar.move();
  myCar.display();
}
```
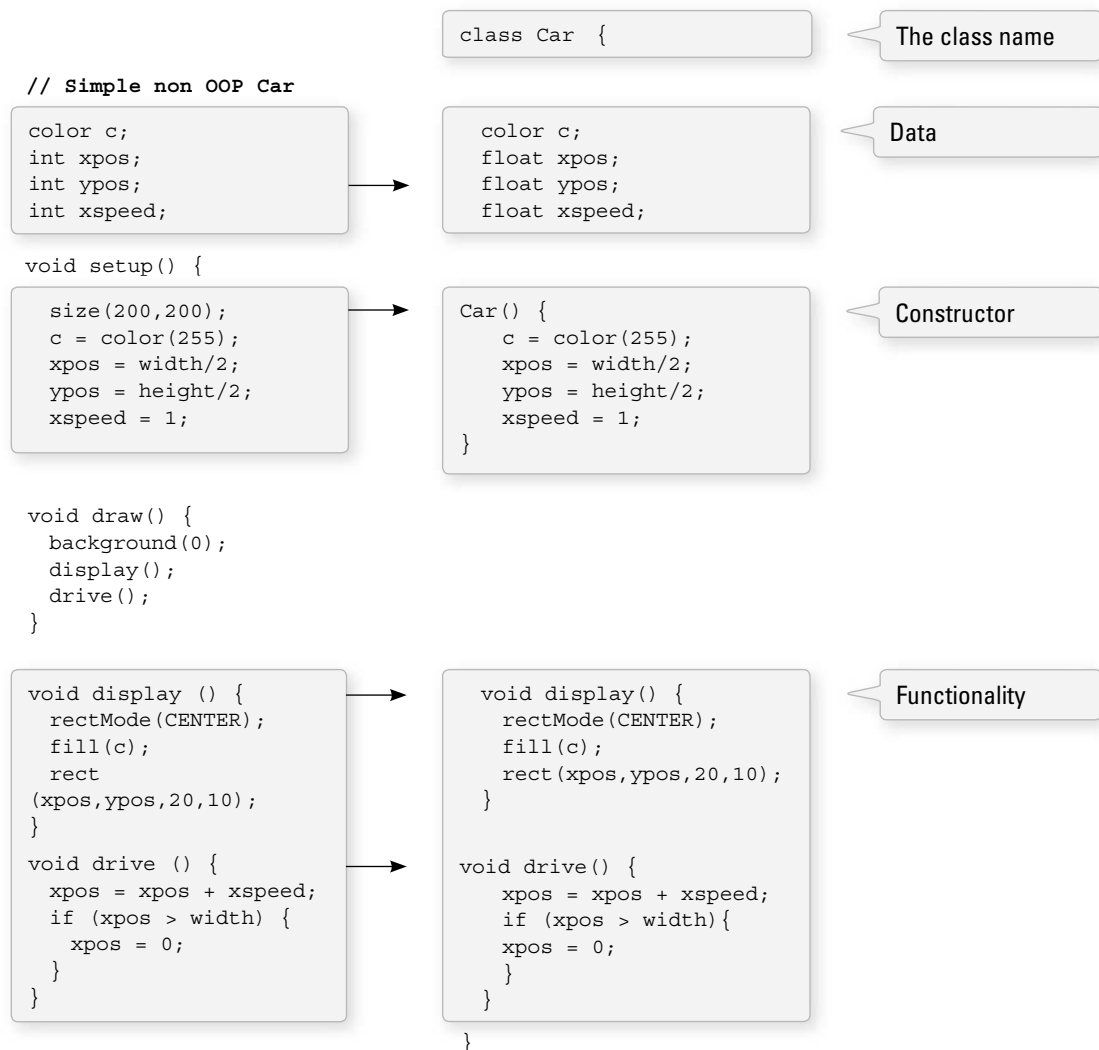
We are going to get into the details regarding the previous code in a moment, but before we do so, let's take a look at how the Car *class* itself is written.

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.

```
class Car {
```
The class name

```
// Simple non OOP Car
```

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

```
color c;
float xpos;
float ypos;
float xspeed;
```
Data

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

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

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

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

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

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

```
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.
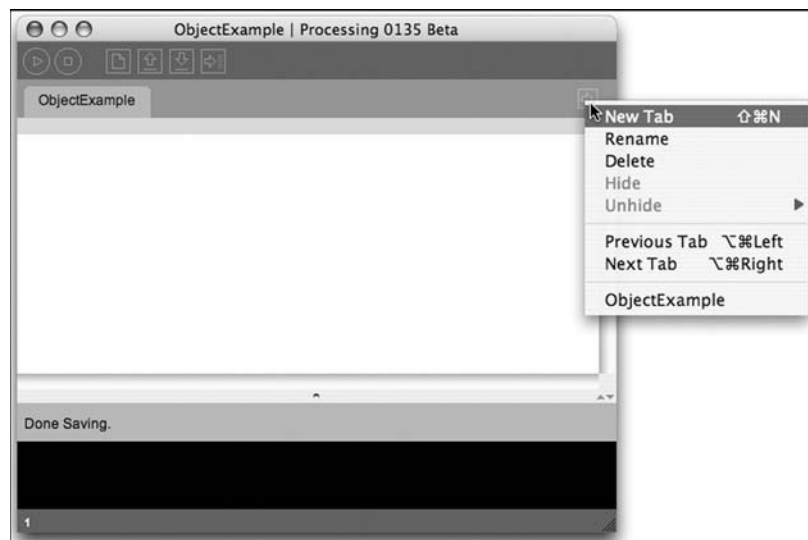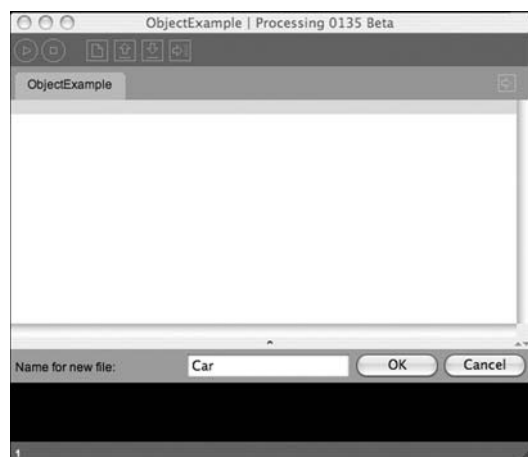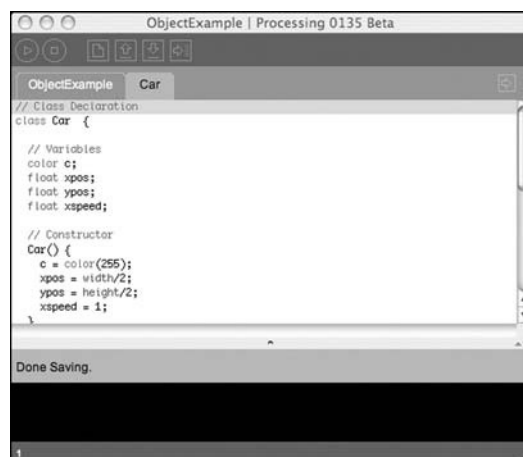
fig. 8.1

fig. 8.2

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.

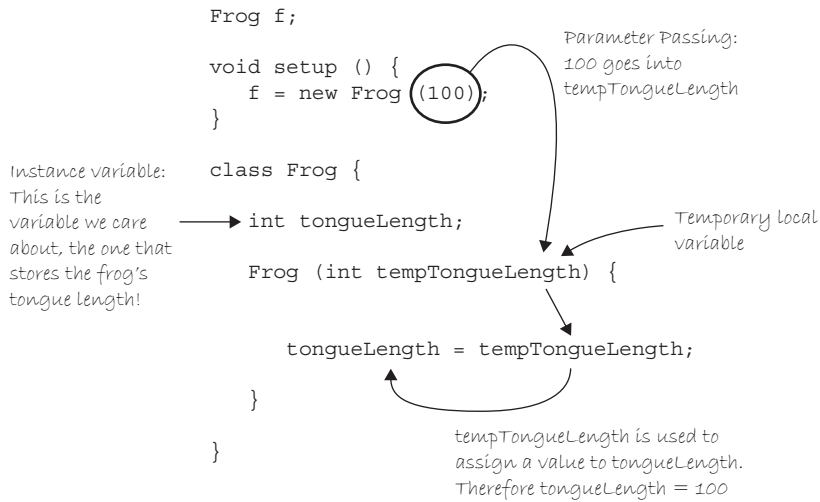We can do this by placing arguments inside of the constructor method.

```
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);
}
```
Parameter Passing:
100 goes into
tempTongueLength

```
class Frog {
```
Instance variable:
This is the
variable we care
about, the one that
stores the frog's
tongue length!

```
    int tongueLength;
```
Temporary local
variable
```
    Frog (int tempTongueLength) {

        tongueLength = tempTongueLength;

    }

}
```
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
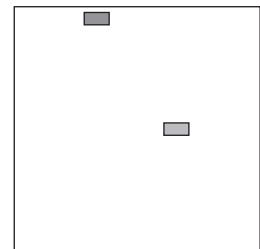when the object is constructed.

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;            A class can include other objects among its variables.
  Spoon spoon;

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

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) {      A function can have an object as its argument.
// Code to determine if
// the bullet struck the enemy
}
```

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;
```
> Zoog is an object!

```
void setup() {
  size(200,200);
  smooth();
  zoog = new Zoog(100,125,60,60,16);
}
```
> Zoog is given initial properties via the constructor.

```
void draw() {
  background(255);
  // mouseX position determines speed factor
  float factor = constrain(mouseX/10,0,5);
  zoog.jiggle(factor);
  zoog.display();
}
```
> Zoog can do stuff with functions!

```
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;
  }
```
> 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);

  }
}
```
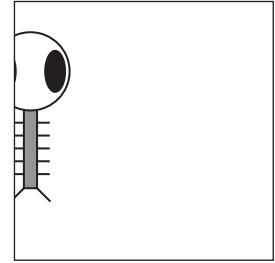

fig. 8.7

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.