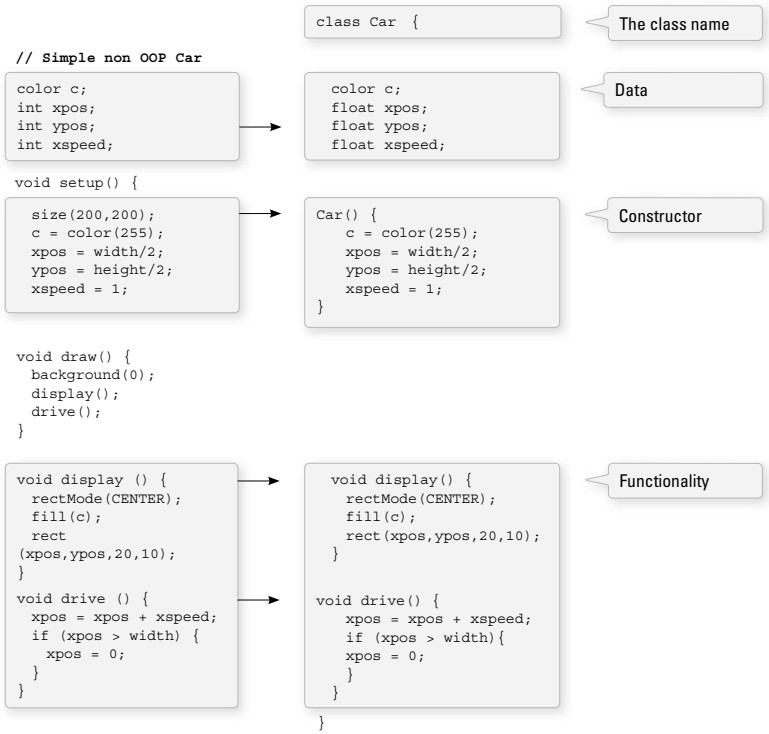
8.3 Writing the Cookie Cutter

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.



- **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 {

}
```

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".

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 {

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

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

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

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

Define a class below the rest of the program.

Variables.

A constructor.

Function.

Function.

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.

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").

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.

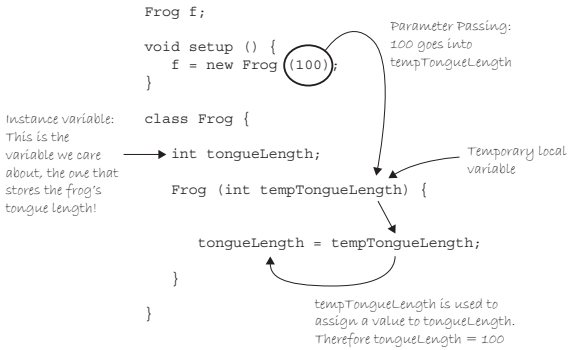
```
Car myCar = new Car(color(255,0,0),0,100,2);
```

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.



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.

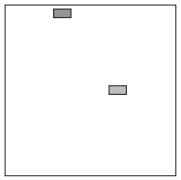


fig. 8.6


```
  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.