Object Oriented Programming

This tutorial is for Processing version 1.0+. *If you see any errors or have comments, please let us know.*

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. 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 things an object can do are functions. Object-oriented programming is the marriage of all of the programming fundamentals: data and functionality.

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. Th is 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*.

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.

Consider the pseudo-code for a simple sketch that moves a rectangle horizontally across the window (we'll think of this rectangle as a "car").

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.

To implement the above pseudo-code, we would define global variables at the top of the program, initialized them in setup(), and call functions to move and display the car in draw(). Something like:

```
color c = color(0);
float x = 0;
float y = 100;
float speed = 1;

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

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

void move() {
  x = x + speed;
  if (x > width) {
    x = 0;
  }
}

void display() {
  fill(c);
  rect(x,y,30,10);
}
```

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*. The object will also know about the *stuff it can do*, the methods (functions inside an object) - the car can *drive* 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.
- Drive 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(255);
  myCar.drive();
  myCar.display();
}
```

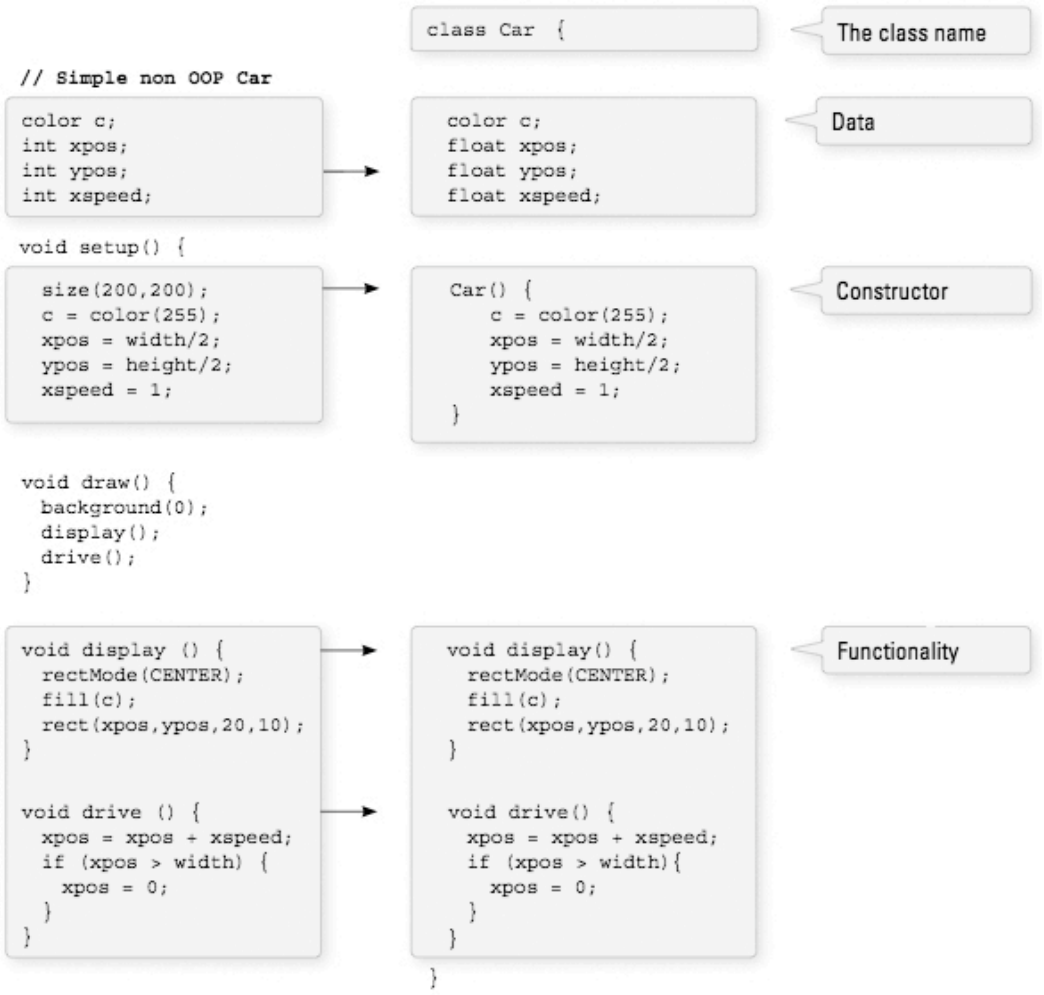
We are going to get into the details regarding the above code in a moment, but before we do so, let's take a look at how the Car class itself is written.

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, recreating exactly the same look and behavior in an object-oriented manner.

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 and place them into a Car class, from which we will then be able to make Car objects.



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.

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.

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

```
void setup() {

}

void draw() {

}

class Car {

}
```

Using an Object: The Details

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

```
// Step 1. Declare an object.
Car myCar;

void setup()  {
  // Step 2. Initialize object.
  myCar = new Car();
}

void draw()  {
  background(255);
  // Step 3. Call methods on the object.
  myCar.drive();
  myCar.display();
}
```

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.

A variable is always declared by specifying a type and a name. With a primitive data type, such as an integer, it looks like this:

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

Primitive data types are singular pieces of information: an integer, a float, a character, etc. Declaring a variable that holds onto an object is quite similar. The diff erence 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.

In order to initialize a variable (i.e., give it a starting value), we use an assignment operation - variable equals something. With a primitive (such as integer), it looks like this:

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

Initializing an object is a bit more complex. Instead of simply assigning it a value, like with 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, 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.

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. Calling a function inside of an object is accomplished via dot syntax: variableName.objectFunction(Function Arguments);

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

```
// Functions are called with the "dot syntax".
myCar.drive();
myCar.display();
```

Constructor Arguments

In the above 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.

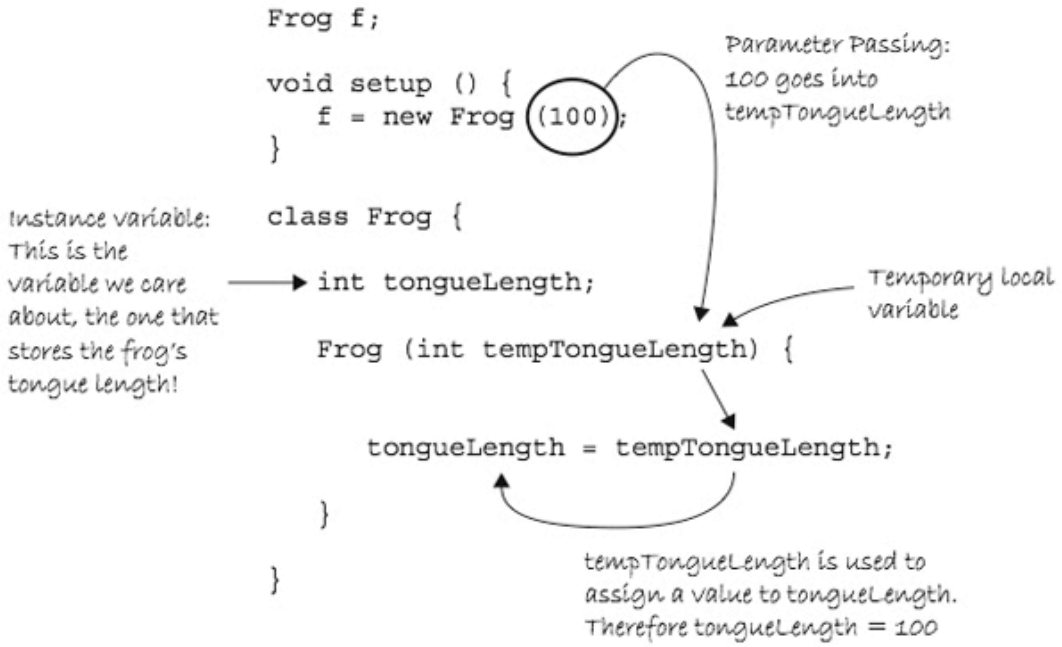
```
Car myCar =  new Car(color(255,0,0),0,100,2);
```

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. Th e code is strange-looking and can seem awfully redundant: "For every single variable I want 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 looks at how parameter works in this context.



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

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 color, 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 sketch with multiple object instances, each with unique properties.

```
// Example: Two Car objects
Car myCar1;
Car myCar2; // Two objects!

void setup() {
  size(200,200);
  // Parameters go inside the parentheses when the object is constructed.
  myCar1 = new Car(color(255,0,0),0,100,2);
  myCar2 = new Car(color(0,0,255),0,10,1);
}

void draw() {
  background(255);
  myCar1.drive();
  myCar1.display();
  myCar2.drive();
  myCar2.display();
}

// 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.
class Car {
  color c;
  float xpos;
  float ypos;
  float xspeed;

  // The Constructor is defined with arguments.
  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 drive() {
    xpos = xpos + xspeed;
    if (xpos > width) {
      xpos = 0;
    }
  }
}
```

Objects are data types too!

Assuming this is your first experience with object-oriented programming, it's important to take it easy. The examples here 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. Th is is perfectly reasonable and quite common in object-oriented programming.

```
class PlaceSetting  {

  Fork fork;
  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)  {
  // Code to determine if
  // the bullet struck the enemy
}
```

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.

Processing was initiated by Ben Fry and Casey Reas. It is developed by a small team of volunteers.