JavaScript for App Development

By Mark Lassoff, Founder Framework Television

Section Seven

Expect this section to step up the complexity considerably. We're moving in to an area that's more ethereal. Move slowly through the material and review until you understand it. Understanding classes and objects is foundational and really undergirds your success in learning to program.

Most of all: Don't give up. Don't let complexity and the work needed to understand it be the reason you don't succeed! I promise that you can do this if you spend the time.

-Mark

We're going to make your world more object oriented.

Object oriented programming came in to favor in the nineties because it provided a systemic way to deal with complex problems that had to be translated in to code. It is, fundamentally, a way of looking at the world that breaks real-world items in to their core properties and actions.

In this section we'll be creating Objects a couple of different ways. First we'll create an object directly. We'll create its properties and initialize them. We'll add it's methods and use them. However the true power of Object Oriented programming will come with our second example—where we'll create a Class, which is a blueprint for an object. We'll then instantiate multiple instances of that class.

Imagine you were tasked with creating a card game. Could you think of anything within the game that could be represented as a class? Perhaps:

- a Card?
- a Hand?
- a Player?

Each of these items could easily be thought of as a Class. A player for example would have a name, bank, and, perhaps and avatar. The value of these properties would differ, but the properties themselves would remain the same.

A card would have a suit and a value. Again, each card would have a different suit and value combination, however, each card would have the suite and value properties.

As with anything, the abstract ideas of classes and objects will better crystalize as you work with them. Let's go!

Section 7 Goals

In this section of the course your goals are:

- ☐ Understand the Design of a Class
- ☐ Create a Simple Class
- ☐ Create a Class Prototype
- ☐ Create Object Instances from That Class

Watch This: Section 7 Video

As always your course videos are available on YouTube, Roku and other locations. However, only those officially enrolled have access to this course

guide, are able to submit assignments, work with the instructor, and get this guide.

Watch this section video at: https://www.youtube.com/watch?v=qxtMtud3f50

Understanding the Design of a Class

Good class design encompasses all of the necessary properties to describe the class and all of the necessary methods to model behavior. The process of boiling a something down to a class with properties and methods is known as abstraction.

Let's take a bank account and through the process of abstraction boil it down to it's properties and methods:

```
Bank Account

Properties:
Account Number
Balance
Account Type
Opening Date
Account Status

Methods:
Open Account
Close Account
Credit
Debit
```

Of course, my abstraction is a slightly simplified version of the real world. Every abstraction is, but, you can see here how with this object we could write could to model an actual bank account.

Do This: Create a Model

Using the bank account example above as a guide, create a class to model a real estate listing. Make it comprehensive. Remember we do not want to include VALUES, just the properties and the methods similar to how the bank account was modeled. We know there will be a balance—but we don't know what the balance is.

> If you are not particularly familiar with the real estate world, visit a site like Zillow and look some listings so you can create your class.

Once you've created your class share your properties and methods using the format above with others in the course via Slack. Note how similar or different your version is from others that are posted.

Creating a Simple Object

So let's take this idea of a class and translate it in to code. This first version of a class we're looking at defines the class and object at the same time. While this has limited utility, it does allow use to create and use and single version of an object.

```
<div id="output"></div>
<script>
   var car = {};
   //Properties
   car.type = "Sedan";
   car.make = "Honda";
   car.model = "Civic";
   car.year = 2015;
   car.weight = 3000;

   //Method
```

```
  car.startCar = function()
  {
     alert("Car is started");
  }

  car.stopCar = function()
  {
     alert("Car is stopped");
  }

  var out = "Car Type: " + car.type;
  out += "<br/>Car Make: " + car.make;
  out += "<br/>Year: " + car.year;

  car.startCar();
  car.stopCar();

  document.getElementById('output').innerHTML = out;
</script>
```

The code above creates a `car` object. The car is described in terms of its properties `type`, `make`, `model`, `year`, and `weight`. In this case you'll notice that these properties are immediately populated with values. Properties use the common dot notation.

It might be helpful to think of properties as adjectives- the elements that describe the class in your model.

This class also has some methods which are behaviors modeled in the class. In this model (simply) modeled are starting the car and stopping the car with the `startCar()` and `stopCar()` methods.

Remember, in this first example we are creating the class and object at the same time, so, right after the methods are defined we can start using the class.

The properties are access for output, again, using the dot notation. The methods are also run using the dot notation.

Do This: Create Your First Class

Using the class outline you created previously for a real estate listing (and any feedback you received over Slack), create the class using the code above as a guide. You should be able to establish and output the value of the properties and create and use at least one method related to the listing.

Create a Class Prototype

Now let's create a class that is just a blue print for objects. You'll notice there are no values assigned to the properties here. The syntax is a bit different because this time we're creating a model or prototype for the class.

As a standard Class Prototypes are created in their own separate files saved as a Javascript file. The file below should optimally have the filename Car.js.

It then needs to be included in any file that uses it like this: `<script src="Car.js">/script>`

Examine the code below.

```
var Car =  function(type, make, model, year){
   //Properties
   this.type = type;
   this.make = make;
   this.model = model;
   this.year = year;
   this.speed = 0;

   //Methods
   Car.prototype.accelerate= function()
   {
      if(this.speed < 100){
         this.speed += 10;
         console.log("Speed is now: " + this.speed);
      } else
      {
         console.log("Top speed reached");
      }
   },

   Car.prototype.brake = function()
   {
      if(this.speed > 10)
      {
         this.speed -= 10;
         console.log("Speed is now: " + this.speed);
      } else
      {
         this.speed = 0;
         console.log("Speed is now: " + this.speed);

      }
   }

}
```

The code begins with a constructor function that is fired every time an object of this class type is made.

The initial values for all of the properties are set. These values are either passed in when the object is created, or, initialized here.

```
var Car =  function(type, make, model, year){
   //Properties
   this.type = type;
   this.make = make;
   this.model = model;
   this.year = year;
   this.speed = 0;
```

You'll also notice repeated use of the keyword `this`. Meant to be self-reflective, `this` refers to the instance created. We are essentially creating properties that live in the instances created, not in the class itself.

To create an instance of this class you'd use something like:

```
var redCar = new Car("sedan", "Ford", "Tauras", "2010");
```

The instance created, `redCar` now has all of the properties and methods of the Car model. If you accessed the value of `redCar.year` the value would be 2010.

The methods, created with the `prototype` keyword, allow the car to accelerate and brake. Note that the speed is tracked by a variable internal to the class called `speed` and is access in the object created from the model above as `redCar.speed` .

Create and use Class Instance

The code above creates a model for the object, but not the object itself. It is, again, simply a blue print to make `Car` objects in JavaScript.

We would actually create the objects using the code below:

```
<script src="model.js"></script>
<script>
   //Create object or instance
   var myHonda = new Car("Sedan","Honda", "Civic", 2017)
   var myLexus = new Car("Sedan", "Lexus", "330", 2015);
   alert(myHonda.model);
   myHonda.accelerate();
   myHonda.accelerate();
   myHonda.accelerate();
   myHonda.accelerate();
   myHonda.brake();

   alert(myLexus.model);
   myLexus.accelerate();
   alert(myLexus.make + " going " + myLexus.speed + " MPH.");
</script>
```

We actually create two distinct objects in this example: `myHonda` and `myLexus`. Each of the objects created is distinct and carries unique property values. The objects are then "exercises" by implementing the methods `accelerate()` and `brake()`.

Do This: Debugging

Assume the correct instantiation for this would be:

```
var sheep = new Animal("mammel", 124, "White", true, 42);
```

This code isn't working.

Filename: Animal.js

```
var Animal =  function(type, weight, color, fur, length){
   //Properties
   this.type = type;
   this.weight = weight;
   this.color = color;
   this.fur = fur;
   this.length = length;
   this.soundItMakes = soundItMakes;

   //Methods
   animal.Prototype.makeSound= function makeSound()
   {
      alert(this.soundItMakes);
   },

   Animal.Prototype.breathe = function()
   {
      alert(this.type + " is " + breathing);
   }

}
```

To test your corrections, you'll have to create a file that includes this script (Animal.js) and then creates an instance of Animal, and implements the `makeSound()` and `breathe()` functions.

Submit This: Lab Exercise

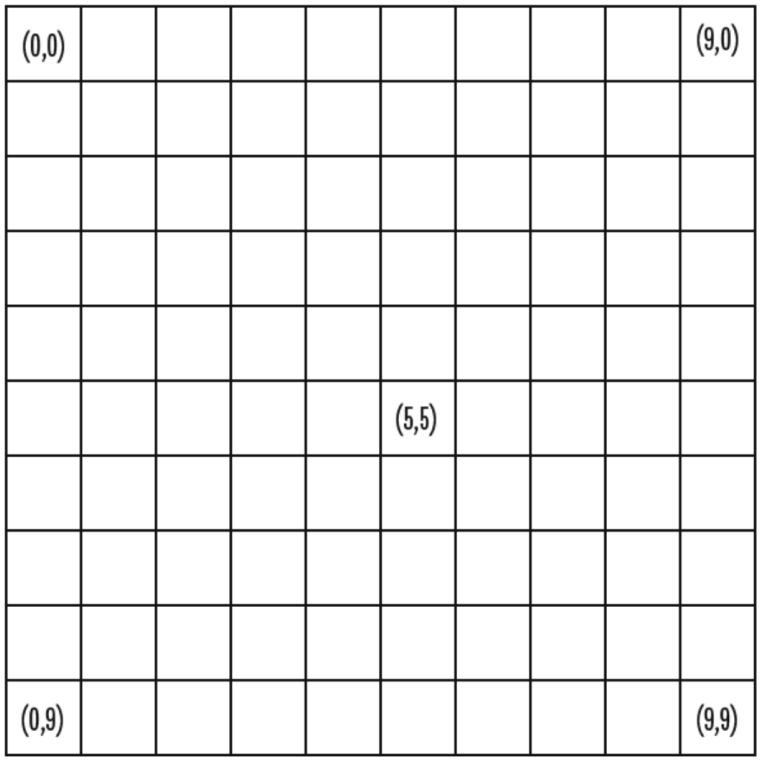
Assume we include the following properties and methods in a prototype of a game player:

Properties:
name (String)
livesLeft (integer)
score (integer)
speed (integer)
gridLocationX (integer)
gridLocationY (integer)

Methods:
die()
moveLeft()
moveRight()
moveUp()
moveDown()

Imagine the game is played on 10 x 10 grid, where the upper left hand corner is position (0,0) and the bottom right is position (9,9). The player Starts at 0,0. `moveRight()` would move the player to (1,0). From the initial starting point `moveDown()) would move the player to (0,1).

The player should not be able to fall off the board.



Knowing what you know create a model for a Player using the properties and methods above. Implement the methods according to the rules in the discussion.

A couple of hints:

- When you initialize the Player in the constructor `gridLocationX` and `gridLocationY` should be initialized to 0.
- X is left to right. Y is up and down.
- So when creating the `moveUp()` method you want to include the following: `if (this.gridLocationY > 0) { this.GridLocationY =- 1 };` meaning we can't have a gridLocationY of less than zero.
- Solve one problem at a time to avoid overwhelm.

Your only deliverable is the model file which should be named `Player.js`.

As mentioned at the beginning, this represents a significant increase in sophistication from previous labs. Expect to struggle with this lab and request help from other members and your instructional team in the class community.

There will be a solution set circulated for this problem.

Good luck!

Please save your file in the following format to insure proper credit:

`LastName_Exercise7.`

Remember every exercise must be submitted in order for you to earn certification. Once you've completed this exercise, you're ready to move on to Section 8.