An Introduction to Object-Oriented JavaScript

aka

OOJS

pronounced *Oh...Jeeze*

Learning Goals

- Students create and use functions with parameters *(mastery)*
- Students organize functions into classes and objects *(functional)*
- Students apply SOLID patterns to JavaScript functions *(mastery)*
- Students make effective use of `this` in multiple contexts *(functional)*

Warm Up

Take 5 minutest to discuss the following with one or two people near you:

- What are the main components of Object Oriented Programming?
- What's one way to create a new object in JavaScript?
- How have you worked to make your JavaScript object-oriented thus far?
- What do you like/dislike about OOP?

Introduction

JavaScript is an object-oriented programming language, but it follows a slightly different approach than what we know from Ruby. Instead of creating classes, constructor functions can be used to construct new objects in JavaScript.

It's not a rule baked into the language, but — by convention — most JavaScript developers capitalize the names of functions that they intend on using as object constructors.

Object Constructors

A constructor function or object constructor can be thought of as a blueprint (similar to classes), or—better yet—as a casting mold from which new objects are minted. The constructor function includes basic information about the properties of an object and uses a special syntax that allows us to build new objects quickly using the template defined by the constructor.

Object constructors can be called using the `new` keyword.

```
function Dog() {}

var fido = new Dog();
```

`fido` in the example above will be a new object — albeit, a very simple one.

Let's add to our `Dog()` constructor.

```
function Dog(name) {
  this.name = name;
  this.legs = 4;
}

var fido = new Dog('Fido');
var spot = new Dog('Spot');

fido.name; // 'Fido'
fido.legs; // 4
spot.name; // 'Spot'
```

Functions and `this` revisited

`Dog` is just a regular function. But, we call it a little differently than we did in previous section on functions. If you recall, there are a few ways we can call a function:

- Using a pair of parenthesis as the end of the functions name (e.g. `someFunction()`).
- Using the `call()` method (e.g. `someFunction.call()`).
- Using the `apply()` method (e.g. `someFunction.apply()`).

When we are writing object-oriented JavaScript, we have a fourth way of invoking a function: the `new` keyword. The `new` keyword invokes the function *as a constructor*, which causes it to behave in a fundamentally different way.

When we use the `new` keyword to call our function as a constructor, a few things happen under the scenes:

1. `this` is set to a new empty object
2. The prototype property of the constructor function (`Dog.prototype` in the example above) is set as the prototype of the new object, which was set to `this` in the first step
3. the body of our function is run
4. our new object, `this`, is returned from the constructor

The `prototype` property

Let's take a look at this in the context of our `Dog()` constructor:

```
function Dog(name) {
  // `this` is set to a new object
  // the prototype is set to `Dog.prototype`
  this.name = name;
  this.legs = 4;
  // return this;
}
```

What is `Dog.prototype` and where does it come from? Functions are objects and all functions in JavaScript have a `prototype` property.

This property is set to an empty object — `{}` — by default.

```
function Dog() {}
function Cat() {}

Dog.prototype; // {}
Cat.prototype; // {}
```

With regular functions, we generally don't use the `prototype` property — it's like an appendix. But, this special little object comes in to play when we use the function as a constructor.

You may have heard that JavaScript has something called *prototypal inheritance*. This is a very complicated term for a relatively simple concept.

When we call a property on an object (e.g. `fido.name`), JavaScript checks the object to see if it has a `name` property.

If it does, then it hands us that property. If not, then it checks the object's prototype.

If the object's prototype doesn't have that property, then it check's the prototype's prototype, and so on.

It continues this process until it reaches the top of the chain. If it still hasn't defined this property, then it returns `undefined`.

By default, all objects inherit from `Object`, which has a few methods on it. One of these methods is `toString()`.

```
function Dog(name) {
  this.name = name;
  this.legs = 4;
}

var fido = new Dog('Fido');

fido.legs; // 4
fido.toString(); // [object Object]
```

When we call `fido.legs` in the example above, JavaScript checks `fido` to see if it has a `legs` property. It does, so JavaScript returns the value, `4`. In the next line, we call `toString()`.

Well, `fido` doesn't have a `toString` property, so we check `fido`'s prototype, which is `Dog.prototype`. That's an empty object, so it certainly doesn't have that property.

Eventually, we work our way up to `Object.prototype`, which has a to `toString` property set to a built-in function. JavaScript calls the `toString()` method that it found up the chain, which returns `[object Object]`.

We could, however, set our own `toString()` method that would return something a little more helpful.

```
function Dog(name) {
  this.name = name;
}

var fido = new Dog('Fido');

fido.toString = function () {
  return '[Dog: ' + this.name + ']';
};

fido.toString(); // [Dog: Fido]
```

JavaScript finds the `toString` property immediately and doesn't have to look up the chain of prototypes. But, only `fido` has this fancy new `toString` property. It would be nice if all dogs could share this new functionality.

Each dog constructed by the `Dog()` constructor has `Dog.prototype` set as its prototype. This means that each dog looks immediately to `Dog.prototype` if we ask for a property that it doesn't have.

Consider the following:

```
function Dog(name) {
  this.name = name;
}

Dog.prototype.toString = function () {
  return '[Dog: ' + this.name + ']';
};

var fido = new Dog('Fido');
var spot = new Dog('Spot');

fido.toString(); // [Dog: Fido]
spot.toString(); // [Dog: Spot]
```

Prototypes are a great way to share functionality between related objects. We can define any properties we want on `Dog.prototype`.

```
function Dog(name) {
  this.name = name;
}

Dog.prototype.sayHello = function () {
  return 'Hello, my name is ' + this.name + '.'
};

var fido = new Dog('Fido');
var spot = new Dog('Spot');

fido.sayHello(); // Hello, my name is Fido.
spot.sayHello(); // Hello, my name is Spot.
```

Here is the same implementation in ES6 syntax:

```
class Dog {
  constructor (name) {
    this.name = name
  }

  sayHello () {
    return `Hello, my name is ${this.name}.`
  }
}

const fido = new Dog('Fido')
const spot = new Dog('Spot')

fido.sayHello() // Hello, my name is Fido.
spot.sayHello() // Hello, my name is Spot.
```

Don't let the `class` keyword fool you **too** much. It still compiles down to a `Dog.prototype` object, it's just wrapped in a container more familiar to other OO lanuages.

A jQuery Example

You are now familiar with the classic `$(document).ready(() => ...)` setup, which tells the browser to wait for the DOM to load before running your scripts. Perhaps you've also ended up with a big ol' list of functions in and out of this `ready` block, not really organized in any object-oriented way.

A potential solution for organizing your jQuery setup looks like this:

```
// some higher-level .js file
$(document).ready(() => {
  new EventHandler()
})

// in a separate "eventHandler.js" file:

// ES5
function EventHandler() {
  $("button").on("click", this.doSomething) // this is "listening" upon its construction
}

EventHandler.prototype.doSomething = function() {
  console.log("Handled!")
}
```

```
// ES6
class EventHandler {
  constructor() {
    $("button").on("click", this.doSomething.bind(this))
  }

  doSomething() {
    console.log("Handled!")
  }
}
```

Your turn: Mod 1 Final Returns!

Go to this repo and follow the instructions to get set up. Let's take thirty minutes to implement (some of) the Mod 1 final using object-oriented JavaScript.

Prototypes vs. Classes

This is a topic that has been hashed out to great length on the internet. In some ways there are a lot of similarities between classes in a language like Ruby and prototypes in a language like JavaScript:

- Both allow child instances of their type to access their methods and behavior
- Both use a "chain" mechanism to continue searching for requested properties in their parent
- Both can be used (via constructor invocation) to "set up" new objects when they are created

There are also some major differences:

- Prototypes don't really provide a mechanism for encapsulation of state, which is one of the major principles of most OO languages.
- JavaScript doesn't provide an OO-style mechanism for "private" functions (although we can achieve something similar with closures).
- Prototypes don't distinguish between their own methods and the methods provided to their children (i.e. class methods vs. instance methods).

Closing

With a neighbor, discuss three things you can do to update your QS code to make it more object oriented.