Object Oriented JavaScript

Procedural Programming

- Most of the programming we've been doing so far has been of the procedural kind, meaning step-by-step.
- In procedural programming, we simply write out exactly what our program is doing in a line-by-line manner
- This makes it difficult to reuse any code we've written, or build interactive code

Object Oriented Programming

- Rather than writing all of our code in one monolithic block, we can organize it into objects
- These objects have attributes (variables) and methods (functions) that can be referenced and changed
- Rather than maintaining all of our functions and variables in a flat structure, we attach them to objects that maintain and encapsulate them

Object Oriented Programming

- Objects have awareness of their own functionality, attributes, and methods.
- Objects, in almost all cases, shouldn't refer to things declared outside of the object. Data should either be managed internally, or passed in via functions.
- This will increase the ease of testing and readability later on in the development process.

The Singleton Pattern

The first and simplest style of object oriented javascript is creating Singletons, individual objects that have data and methods for managing one thing. In this example, once you call `Form.init()`, the Form singleton manages everything else.

```
var Form = {
  hasSubmitted: false,

  init: function() {
    this.username = document.getElementById("username");
    this.password = document.getElementById("password");
    this.submitButton = document.getElementById("submit");

    this.submitButton.addEventListener("click", this.submit.bind(this));
  },

  submit: function() {
    console.log("Submitted form with username " + this.username.value);
    console.log("Your password is " + this.password.value.length + " long");
    this.hasSubmitted = true;
  },
};
```

The Singleton Pattern

- This pattern is good for things that you know you won't need multiple of.
- If you decide to later allow for multiple (i.e. many '`Form`'s in one page) it's easy to refactor.
- Examples of singletons could be: `NavBar`, `SignupForm`, or anything that manages an entire page.

What's `this` refer to?

- The `this` keyword refers to the "owner" of the current context.
- When used in the global context, `this` refers to `window`. Inside an object, `this` refers to that instance of the object. In an event, `this` refers to the element the event is bound to.
- Because of changing contexts, we sometimes have to `bind()` our previous context to a function:

What's `this` refer to? (Example)

```
button.addEventListener("click", function() {
  console.log(this); // Prints out the button element
});

button.addEventListener("click", function() {
  console.log(this); // Prints out what `this` was in previous context
}.bind(this));
```

What's `this` refer to? (Example)

```
var Singleton = {
  init: function() {
    this.button = document.getElementById("button");

    this.button.addEventListener("click", function() {
      // Would error out with "button.doThing is not a function"
      // if we hadn't bound Singleton's this to the function.
      this.doThing();
    }.bind(this));
  },

  doThing: function() {
    alert("You did a thing!");
  },
};
```

Exercise: Singletons

- Create a singleton object that has some properties (data) and try printing them out using dot notation (Like `Form.isSubmitted`)
- Add a function to your singleton, and try calling it from outside of the singleton (Like `Form.init()`)
- Add another function to your singleton, and have your first function call the new one using the `this` keyword.
- Finally, make a function that binds a callback to an event, and have that callback call a function on your singleton. Don't forget to `bind(this)` the callback!
- You can bind to `window.addEventListener("resize")` if you don't want to add an element.

Classes

Classes are for when we need multiple instances of the same type of object. The class is the blueprint, and instances of it are items made to the spec of the blueprint.

```
class AlertButton {
  constructor(element, text) {
    this.element = element;
    this.text = text;

    this.element.addEventListener("click", this.handleClick().bind(this));
  },

  handleClick() {
    alert(this.text);
    this.beenClicked = true;
  },
}

var greetingBtn = new AlertButton(someButtonElement, "Hello!");
var farewellBtn = new AlertButton(otherButtonElement, "Goodbye!");
```

Classes

- Classes should be flexible, reusable, and self-contained.
- There can be as many instances of a single class as you want. You create a new instance by using the `new` keyword.
- They cannot have properties statically defined, must be set in `constructor`.
- Other examples of where we would use classes are popup modals, slideshows, form elements.

Difference between classes and their instances

- When we refer to a class, we refer to the abstract definition of it
- For example, a `Car` class can have `brand` and `wheelCount` properties
- When we refer to an **instance** of a class, we're talking about a specific object created from that class
- For example, an instance of the `Car` class has a `brand` of `"toyota"` and a `wheelCount` of `4`

Exercise: Classes

- Create a new class of your choosing. Have it accept at least one argument in its constructor.
- Instantiate 3 objects using your new class.
- Add an argument to the constructor, and set a property on the instance using the `this` keyword. Try logging the property on all 3 of your instances.
- Define a method on the class. Have it do something with the property (Double a number, all caps a string etc.) and log what it did to the console. Call this method on all of your instances.

Class Inheritance

- When you make flexible and reusable classes, you'll often times want to specialize them for certain uses
- With inheritance, we can inherit the properties and methods of a parent class, and add or change functionality in the child class by extending it
- Everything that was defined in the parent is usable and overwritable by the child class

Example: Class Inheritance

```
class Vehicle {
  constructor(wheelCount) {
    this.wheelCount = wheelCount;
    this.fuel = 100;
    this.speed = 0;
  }

  drive() {
    this.speed = 60;
    this.fuel--;
  }

  brake() {
    this.speed = 0;
  }
}
```

Example: Class Inheritance

```
class Car extends Vehicle {
  constructor(passengers) {
    super(4);
    this.windowState = "up";
  }

  rollWindowsUp() {
    this.windowState = "down";
  }

  rollWindowsDown() {
    this.windowState = "up";
  }
}
```

Example: Class Inheritance

```
class Motorcycle extends Vehicle {
  constructor() {
    super(2);
    this.isWheelieing = false;
  }

  // Override Vehicle's drive() method
  drive() {
    super.drive();
    this.speed = 40;
  }

  wheelie() {
    this.isWheelieing = true;
  }
}
```

Exercise: Class Inheritance

- Make a new class, but inherit everything from the class you made in the previous exercise, and make an instance of it
- Add a completely new method to your new class, and call it on your instance
- Override the method that previously did something to a property and logged it, and make it log something entirely different

Alternative "classes"

- The technique and syntax shown previously is a newer standard in Javascript that makes classes easier
- This standard wasn't around until ~2015, so you will often see people doing it the old way
- For this class, you should be using `classes`, but you should be familiar with the other ways too

Alternative "classes"

Constructor Functions

```
function Vehicle(wheelCount) {
  this.wheelCount = wheelcount;
  this.fuel = 100;
  this.speed = 0;
}

Vehicle.prototype.drive = function() {
  this.speed = 60;
  this.fuel--;
}

var instance = new Vehicle(4);
```

Alternative "classes"

Factory Functions

```
function MakeVehicle(wheelCount) {
  var vehicle = {
    wheelCount: wheelCount,
    fuel: 100,
    speed: 0,

    drive: function() {
      this.speed = 60;
      this.fuel--;
    },
  };

  return vehicle;
}

const instance = MakeVehicle(4);
```

Best Practices

Multi parameter objects

```
class Button {
  // BAD.
  constructor(text, color, size) { /*code here*/ }

  // Good!
  constructor(params) { /*code here*/ }
}

// BAD.
var button2 = new Button("Hello", null, "blue");

// Good!
var button1 = new Button({
  text: "Hello",
  color: "blue",
});
```

Indicate private members

- There are ways of making things completely private, but that can be cumbersome for debugging
- If you have properties and methods that shouldn't be used externally, start it off with an underscore
- This is common practice to indicate external things shouldn't use it, so don't use them outside of the object!

```
var Greeter = {
  _numMessagesPrinted: 0,

  sayHello(person) {
    var message = "Hello, " + person;
    this._printMessage(this.message);
  },

  sayGoodbye(person) {
    var message = "Goodbye, " + person;
    this._printMessage(this.message);
  },

  _printMessage(message) {
    console.log(message);
    this._numMessagesPrinted++;
  },
}
```

Resources

- Object Oriented Javascript with ES6 Classes - There's a lot more in here than what I talked about, but don't be afraid!
- Singleton in Javascript by Minko Gechev. Dives a lot deeper in to the singleton pattern.
- Sample Class Code from the Google Chrome Team