**JS OOP**

**Overview**

**What is Object Oriented Programming?**

**OOP (Object Oriented Programming)**is something that has revolutionized the way that programming is done today. If used correctly, it can save you loads of time. It will also help you to avoid repeating code to solve the same simple problem and make maintaining your code easy and effective. In this chapter, you'll learn how to use and implement OOP.

Understanding object-oriented programming principles is the key to making your code more organized and modular. This chapter is designed to teach object-oriented programming principles to help you better understand how to organize your code in the future.

Now, Javascript is not a true OOP language but we can tweak Javascript to make it work and look like other traditional OOP languages.

# OOP in ES5 and ES6

Remember that in PHP, you created a Class as follows:

*class* MyFirstClass

{

public $name = "";

public $property = "I am the first property! Woohoo";

private $secret ="keyword operator from-rainbow">= "Hacker";

public *function* \_\_construct($name)

{

$this->name = $name;

echo "I get called for each instance of this class!";

}

public *function* update\_property($property)

{

$this->property = $property;

}

private *function* update\_secret()

{

$this->secret = "Hero";

}

}

$obj1 = new MyFirstClass("J1"); // runs the \_\_construct function immediately

$obj2 = new MyFirstClass("J2"); // runs the \_\_construct function immediately

You should already be familiar with how to create and use a constructor, how to create a public/private function.

## Javascript ES5

In Javascript ES5, you would create a class like above as follows:

*function* MyFirstClass(name){

this.name = name; //put all public attributes with this.

this.property = "I am the first property! Woohoo"; //put all public attributes with this.

var secret = "Hacker"; //this is a private variable

console.log('I get called for each instance of this class!');

//this is a public method

this.update\_property = function(property){

this.property = property;

}

//put any private method in a variable

var update\_secret = function(){

secret = "Hero";

}

}

let obj1 = new MyFirstClass("J1");

let obj2 = new MyFirstClass("J2");

Note that there is no built in constructor!  Whenever the function is called, it executes everything inside the function, meaning a constructor is not really needed when using ES5.

## Javascript ES6

In Javascript ES6, now you can actually use the word Class!  However, a private variable NOR private methods are not really supported in ES6 (see [this thread for more info](https://stackoverflow.com/questions/22156326/private-properties-in-javascript-es6-classes/52237988#52237988) this thread for more info if you're interested).  If we were to convert all the private methods to just public methods, in ES6, the class above would look as follows.

class MyFirstClass {

//ES6 has a constructor support!

constructor(name) {

this.name = name;

this.property = "I am the first property! Woohoo";

this.secret = "Hacker"; //again in ES6, private variables are not really supported

copy

console.log('I get called for each instance of this class!');

}

//public method

update\_property(property){

this.property = property;

}

//another public method

update\_secret(){

this.secret = "Hero"; //again no private methods nor private variables for ES6

}

}

let obj1 = new MyFirstClass("J1");

let obj2 = new MyFirstClass("J2");

# Prototypes

Note that in ES5, we could create instance methods by using this.{{name of the function}}.  For example, for a class Circle, you may do:

*function* Circle(size){

this.size = size;

this.updateSize = function(new\_size){

this.size = new\_size;

}

}

let circle1 = new Circle(5);

If you've created 100,000 circles, it turns out that for each of the object/circle, a unique updateSize function lives inside each of the 100,000 objects!  Notice how much memory this would consume!

Therefore, for ES5, when instance methods are created, you could use the following convention instead.

*function* Circle(size){

this.size = size;

}

Circle.prototype.updateSize = function(new\_size) {

this.size = new\_size;

}

let circle1 = new Circle(5);

The two codes above are exactly identical!  However, adding the method updateSize in the prototype is more memory efficient as the updateSize method only lives inside a single prototype for the entire function Circle.  Both are doing identical task and most of the time in Javascript, if you're not creating lots of instances of the class, I personally prefer doing it the first way.

With ES6, you don't need to do this trick of adding the method in the prototype and adding an instance method automatically adds the method in the prototype for you.

For more information about prototype, please spend a few minutes reading this article: <https://www.w3schools.com/js/js_object_prototypes.asp>

## Assignment: Bike

Create a new class/function called **Bike** with the following properties/attributes:

* price
* max\_speed
* miles

Create 3 instances of this bike.

Now add a constructor method to the class (if using ES6) and require the user to specify the price and max\_speed of each instance. In the constructor also specify in the code so that the initial miles is set to be 0 whenever a new instance is created.

Add the following functions for this class:

* **displayInfo()** - have this method display the bike's price, maximum speed, and the total miles driven.
* **drive()**- have it display "Driving" on the screen and increase the total miles driven by 10.
* **reverse() -** have it display "Reversing" on the screen and decrease the total miles driven by 5.

Have the first instance drive three times, reverse once, and have it displayInfo().

Have the second instance drive twice, reverse twice, and have it displayInfo().

Have the third instance reverse three times and displayInfo().

What would you do to prevent the instance from having negative miles?

====

Do this assignment first using ES6.  Then do it using ES5 (without using prototype).  Then do it again using ES5 but with prototype.

**Going back in time**

So how did Javascript developers create objects before ES5?  Remember that before ES5, there must have been ES4, ES3, and so forth.  Also, remember that Javascript is not really a OOP language?

Well.  How did developers create Javascript objects?  You actually have done this before. :)

For example, please take a look below.  Isn't this another way to create a Javascript object where you basically return a Javascript object that starts with { and ends with }?

For example, take a look at this code:

*function* Desk(name) {

var obj = {};

obj.name = name;

obj.x = 0;

obj.y = 0;

obj.color = "black";

obj.mov = function(x, y) {

this.x = x;

this.y = y;

 }

obj.updateColor = function(new\_color) {

this.color = new\_color;

}

return obj;

}

var desk1 = Desk("oak desk");

var desk2 = Desk("maple desk");

desk1.updateColor("brown");

Above is a perfectly valid way of creating a Javascript object with attributes and methods!  Note that above didn't really use any classes or OOP although it has some similar look.

Now, what we would like you to do is to convert the code above but to do so using three different syntaxes.  For example:

1. Use ES5 Syntax - where methods are directly added inside the function
2. Use ES5 Syntax but this time using prototype to add methods
3. Use ES6 Class syntax

This exercise will help you see how all these four methods are all valid ways of creating a Javascript object.

**JS Circles in ES5/6**

Remember the JS circle assignment you did?  Now, update your code with three different versions.

1. Version 1: where you're using ES5 but NOT prototype
2. Version 2: where you're using ES5 with prototype
3. Version 3: where you're using ES6

These repetitions are designed to help you get familiar with all the various ways of creating a Javascript object.

**cLibrary**

Create a javascript library and put it in a file called 'circle.js'.  Your task is to create a circle library (which we will call cLib going forward).

The goal is to structure your library such that anyone can import your library and draw circles by doing something like this:

let cLib = new Circles(100);

cLib.draw\_circles("canvas");

By doing this, your library would create 100 circles (of varying sizes and colors) and draw these circles in a html tag that has the id specified in the draw\_circles method.  Each circle would slowly expand and once it reaches a certain size, it would disappear from the page. For example, consider the following code:

<html>

<head>

<title>JS Circle Demo</title>

<script src="circle.js"></script>

</head>

<body>

<div id='canvas'></div>

<script>

// initialize cLib (circleLibrary) with 100 circles

let cLib = new Circles(100);

// draw 100 circles on #canvas

cLib.draw\_circles("canvas");

// any other code you need to make this work

</script>

</body>

Create a few versions of this library:

* First, do this using ES5.  Do NOT use a class but use a function.  Do NOT use a prototype.
* Second, do this using a function in ES5 but where you're using a Prototype.
* Third, do this using ES6 where you're creating Class.

This exercise is to get you familiar with all variations of JS syntaxes and also get you familiar with how to create interactive apps like this.

Be creative and make this look really beautiful and creative.

# Inheritance

Remember that all OOP should have inheritance.  How is this done in Javascript?

Typically, the way it's done is as follows:

## ES5 Inheritance

*function* Shape(x, y) {

this.x = x;

this,y = y;

}

*function* Circle(x, y, r) {

Shape.call(this, x, y);

this.r = r;

this.area = function() {

return this.r \* 2 \* Math.PI;

}

}

For ES5, there are several methods to mimic the behavior of inheritance.  Above is just one of the common ways but there are multiple other ways to mimic the behavior of a inheritance.  Note that what ES5 is trying to do is to construct a function such that it mimics the behavior of OOP inheritance (again as Javascript is really not built to handle traditional OOP syntaxes).  Note that the instance methods could either be inside the function or be pulled out separately like this:

*function* Shape(x, y) {

this.x = x;

this,y = y;

}

*function* Circle(x, y, r) {

Shape.call(this, x, y);

this.r = r;

}

Circle.prototype.area = function() {

return this.r \* 2 \* Math.PI;

}

## ES6 Inheritance

class Shape {

constructor(x, y) {

this.x = x;

this.y = y;

}

}

class Circle extends Shape {

constructor(x, y, r) {

super(x, y);

this.r = r;

}

area() {

return this.r \* 2 \* Math\*PI;

}

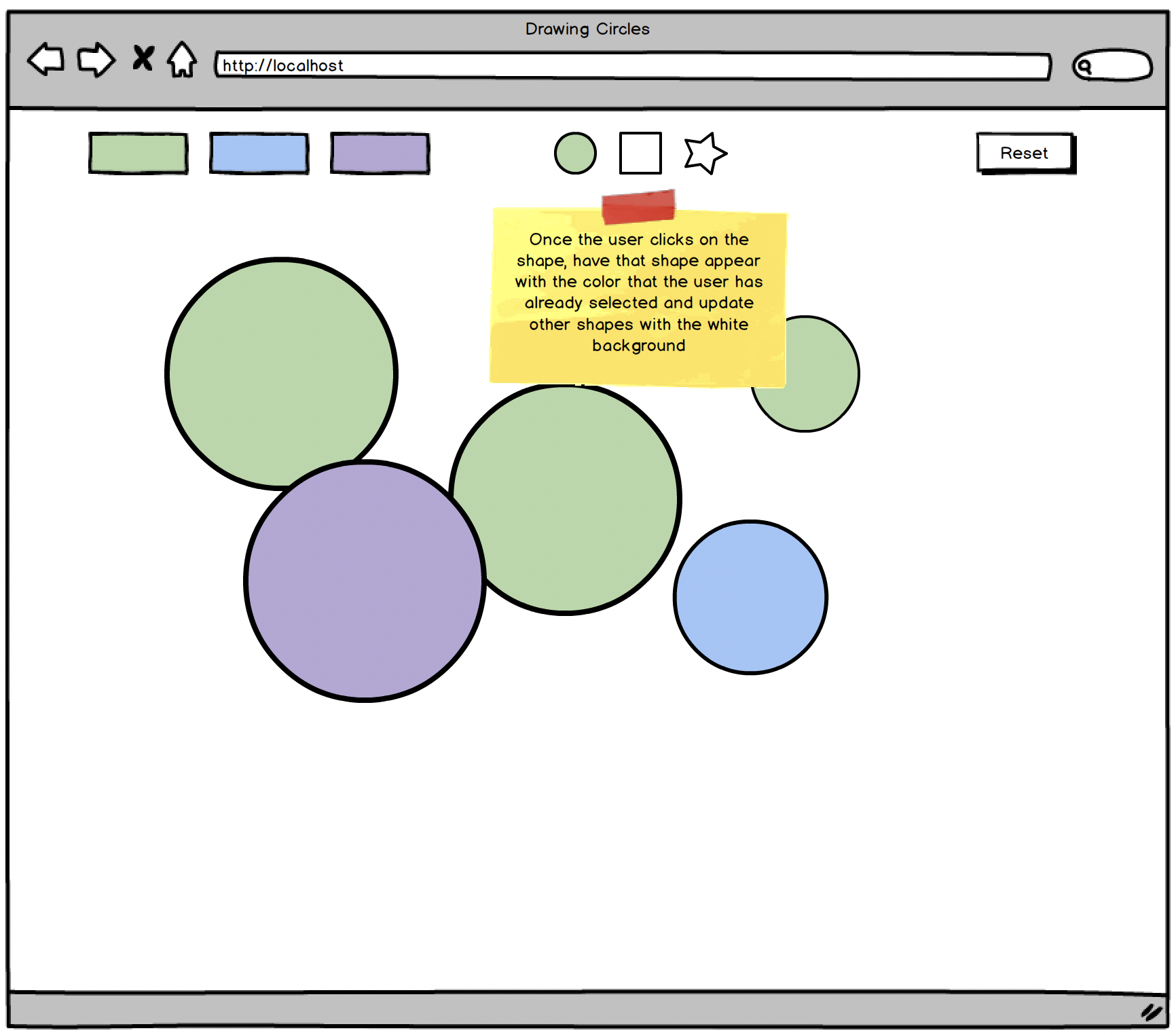
}

Now how ES6 makes it look a lot more like other OOP languages where the class can be '**extended**'.  Note also how you can call the parent class by using **super**.

# JS Circles with Shapes

Now that you've learned about Inheritances, let's put them into use!

For the JS Circles assignment, using the ES6 version, update it so that now it supports multiple shapes!  Look at the UI below.



Note that by default, the green color and the circle should be selected.  Whatever shape is selected, it should show that shape with a color that was selected by the user (or selected by default) and show other shapes with the white. background.  This will help the user know visually which shape was already selected (white background shape would indicate that shape wasn't selected).

Use ES6 inheritance and instead of having a Circle class, it probably makes more sense to have a parent class that Circle, Rectangle, and Star can also inherit from.

START WORKING ON THIS

**Deck of Cards**

Now that you are more familiar with the JS syntaxes, let's have you create a program that's a bit more complex.  Using ES6 syntaxes, please accomplish the following exercise:

Create a Card class. A card should have the following functionality:

* Each Card should have a suit ("Hearts", "Clubs", "Diamonds", "Spades")
* Each Card should have a string value (eg, "Ace", "Two", ...., "Queen", "King")
* Each Card should have a numerical value (1-13)
* Each Card should have a show method (log the card's information to the console)

Create a Deck class. A deck should have the following functionality:

* The Deck should contain the 52 standard Cards
* The Deck should be able to [shuffle](https://bost.ocks.org/mike/shuffle/)
* The Deck should be able to reset
* The Deck should be able to deal a random Card
  + Deal should return the Card that was dealt and remove it from the Dec