URL**: http://www.newthinktank.com/2015/09/object-oriented-javascript/**

<html lang="en">

<head>

<meta charset="UTF-8">

<title>Object Oriented JavaScript</title>

</head>

<body>

<script type="text/javascript">

**// A basic JavaScript object with properties and a method**

var customer = {

name: 'Tom Smith',

speak: function(){

// this allows you to reference a specific objects value

// without knowing the objects name

return "My name is " + this.name;

},

// Objects can contain other objects

address: {

street: '123 Main St',

city: 'Pittsburgh',

state: 'PA'

}

};

document.write(customer.speak()+ "<br />");

// You access properties and object properties like this

document.write(customer.name + " lives at " + customer.address.street + "<br />");

// You can add properties

customer.address.country = "US";

document.write(customer.address.country + "<br />");

// Creating multiple objects of the same type with Constructor

// Functions. Constructors provide the functions that classes

// provide in other languages

function Person(name, street) {

// this allows us to refer to an object even though we

// don't know its name before it is created

this.name = name;

this.street = street;

this.info = function(){

return "My name is " + this.name + " and I live on " + this.street;

}

}

// You call constructor functions with new

var bobSmith = new Person("Bob Smith", "234 Main St");

document.write(bobSmith.info() + "<br />");

// instanceof tells you if an object is of a certain type

document.write("Bob is a person : " + (bobSmith instanceof Person) + "<br />");

// You can pass an object to a function and change values

function changeName(person){

person.name = "Sue Smith";

}

changeName(bobSmith);

document.write("Bob became " + bobSmith.name + "<br />");

// Objects are only equal if they reference the same object

var person1 = new Person("Paul", "123 Main");

var person2 = new Person("Paul", "123 Main");

document.write("Are they equal " + (person1 == person2) + "<br />");

// ---------- PROTOTYPE ----------

// Every function has a prototype property that contains an object

// You can add properties and methods to the prototype object

// and then when you call for them to execute they are used

// just as if they belonged to the object

function getSum(num1, num2){

return num1 + num2;

}

// Get the number of function arguments

document.write("Num of arguments : " + getSum.length + "<br />");

// You can add properties and methods to this object

function Mammal(name){

this.name = name;

this.getInfo = function(){

return "The mammals name is " + this.name;

}

}

// Use prototype to add a property

Mammal.prototype.sound = "Grrrrr";

// Use it to add a method

Mammal.prototype.makeSound = function() {

return this.name + " says " + this.sound;

};

var grover = new Mammal("Grover");

document.write(grover.makeSound() + "<br />");

// List all properties of an object

for( var prop in grover){

document.write(prop + " : " + grover[prop] + "<br />");

}

// Check which property belongs to prototype vs. the object grover

document.write("name Property of Grover : " + grover.hasOwnProperty("name") + "<br />");

document.write("sound Property of Grover : " + grover.hasOwnProperty("sound") + "<br />");

// You can add methods to built in JS objects

Array.prototype.inArray = function inArray(value){

for(i = 0; i < this.length; i++){

if(this[i] === value){

return true;

}

}

return false;

}

var sampArray = [1,2,3,4,5];

document.write("3 in array : " + sampArray.inArray(3) + "<br />");

// ---------- PRIVATE PROPERTIES ----------

// All properties in an object are public in that any function

// can modify or delete these properties.

// You can make properties private by declaring them as

// variables in a constructor

function SecretCode(){

// This value can't be accessed directly

var secretNum = 78;

// This function can access secretNum

this.guessNum = function(num){

if(num > 78){

return "Lower";

} else if(num < 78){

return "Higher";

} else {

return "You Guessed It";

}

}

}

var secret = new SecretCode();

// Returns undefined

document.write("Value of secretNum : " + secret.secretNum + "<br />");

document.write("Is 70 the number : " + secret.guessNum(70) + "<br />");

// Even if we add another function it can't access the secretNum

SecretCode.prototype.getSecret = function(){

return this.secretNum;

}

document.write("The secret number is " + secret.getSecret() + "<br />");

// ---------- GETTERS AND SETTERS ----------

// Getters and Setters can protect data, or provide useful

// ways to set its value

// I'll show you a bunch of getters and setters you may come

// in contact with

var address = {

street: "No Street",

city: "No City",

state: "No State",

// Provides styled data all at once

get getAddress() {

return this.street + ", " + this.city + ", " + this.state;

},

// Allows the user to set 3 values with 1

set setAddress (theAddress) {

var parts = theAddress.toString().split(", ");

this.street = parts[0] || '';

this.city = parts[1] || '';

this.state = parts[2] || '';

}

}

address.setAddress = "123 Main St, Pittsburgh, PA";

document.write("Address : " + address.getAddress + "<br />");

document.write("City : " + address.city + "<br />");

// ---------- CONSTRUCTOR GETTERS AND SETTERS ----------

// Still used even though it is (Deprecated)

function Coordinates(){

this.latitude = 0.0;

this.longitude = 0.0;

}

// Define the getter with the prototype to assign it to with

// the property name and the getter function

Object.\_\_defineGetter\_\_.call(Coordinates.prototype, "getCoords", function(){

return "Lat : " + this.latitude + " Long: " + this.longitude;

});

// Define the setter with the prototype to assign it to with

// the property name and the setter function

Object.\_\_defineSetter\_\_.call(Coordinates.prototype, "setCoords", function(coords){

var parts = coords.toString().split(", ");

this.latitude = parts[0] || '';

this.longitude = parts[1] || '';

});

var testCoords = new Coordinates();

testCoords.setCoords = "40.71, 74.00";

document.write("Coordinates : " + testCoords.getCoords + "<br />");

// ---------- GETTERS AND SETTERS WITH DEFINEPROPERTY ----------

function Point(){

this.xPos = 0;

this.yPos = 0;

}

// Use defineProperty to set getters and setters

// Pass the prototype to attach to along with the property name

// and define the functions to associate with get and set

Object.defineProperty(Point.prototype, "pointPos", {

get: function(){

return "X: " + this.xPos + " Y: " + this.yPos;

},

set: function(thePoint){

var parts = thePoint.toString().split(", ");

this.xPos = parts[0] || '';

this.yPos = parts[1] || '';

}

});

var aPoint = new Point();

aPoint.pointPos = "100, 200";

document.write("Point Position : " + aPoint.pointPos + "<br />");

// ---------- ECMASCRIPT 5.1 GETTERS AND SETTERS ----------

var Circle = function (radius) {

this.\_radius = radius;

};

Circle.prototype = {

set radius(radius) { this.\_radius = radius; },

get radius() { return this.\_radius; },

get area() { return Math.PI \* (this.\_radius \* this.\_radius); }

};

var circ = new Circle(10);

circ.radius = 15;

document.write("A circle with radius " + circ.radius + " has an area of " + circ.area.toFixed(2) + "<br />");

// ---------- INHERITANCE ----------

// When we ask for a property if it isn't found in the main object

// then it is searched for in the prototype object. We are able

// to inherit methods and variables from any object in a

// chain of objects.

function Animal(){

this.name = "Animal";

// toString is a function in the main Object that every

// object inherits from

this.toString = function() {

return "My name is : " + this.name;

};

}

function Canine(){

this.name = "Canine";

}

function Wolf(){

this.name = "Wolf";

}

// Overwrite the prototype for Canine and Wolf

Canine.prototype = new Animal();

Wolf.prototype = new Canine();

// After you overwrite prototype its constructor points to the

// main object object so you have to reset the constructor after

Canine.prototype.constructor = Canine;

Wolf.prototype.constructor = Wolf;

var arcticWolf = new Wolf();

// Wolf inherits toString from Animal

document.write(arcticWolf.toString() + "<br />");

document.write("Wolf instance of Animal : " + (arcticWolf instanceof Animal) + "<br />");

// Properties added to any object in the chain is inherited

Animal.prototype.sound = "Grrrrr";

Animal.prototype.getSound = function(){

return this.name + " says " + this.sound;

}

Canine.prototype.sound = "Woof";

Wolf.prototype.sound = "Grrrr Wooof";

document.write(arcticWolf.getSound() + "<br />");

// More often then not it makes more sense to just inherit the

// prototype to speed up the lookup process

function Rodent(){

this.name = "Rodent";

}

function Rat(){

this.name = "Rat";

}

Rodent.prototype = new Animal();

Rat.prototype = Rodent.prototype;

Rodent.prototype.constructor = Rodent;

Rat.prototype.constructor = Rat;

var caneRat = new Rat();

// Wolf inherits toString from Animal

document.write(caneRat.toString() + "<br />");

// ---------- INTERMEDIATE FUNCTION INHERITANCE ----------

function extend(Child, Parent){

var Temp = function(){};

Temp.prototype = Parent.prototype;

Child.prototype = new Temp();

Child.prototype.constructor = Child;

}

function Deer(){

this.name = "Deer";

this.sound = "Snort";

}

extend(Deer, Animal);

var elk = new Deer();

document.write(elk.getSound() + "<br />");

// ---------- CALL PARENT METHODS ----------

function Vehicle(name) {

this.name = "Vehicle"

}

// Functions for the parent object

Vehicle.prototype = {

drive: function(){

return this.name + " drives forward";

},

stop: function(){

return this.name + " stops";

}

}

function Truck(name) {

this.name = name

}

// Inherit from Vehicle

Truck.prototype = new Vehicle();

Truck.prototype.constructor = Truck;

// Overwrite drive parent method

Truck.prototype.drive = function(){

// Call the parent method with apply so that the parent

// method can access the Trucks name value

var driveMsg = Vehicle.prototype.drive.apply(this);

return driveMsg += " through a field";

}

var jeep = new Truck("Jeep");

document.write(jeep.drive() + "<br />");

document.write(jeep.stop() + "<br />");

// ---------- SINGLETON PATTERN ----------

// Singletons are used when you only ever want 1 object to

// be created

// Let's say you want to create a game character with fixed

// stats

function Hero(name){

// Check if the object exists

if(typeof Hero.instance === 'object'){

// If it does return it

return Hero.instance;

}

// if it doesn't then create the hero

this.name = name;

Hero.instance = this;

return this;

}

var derekHero = new Hero("Derek");

document.write("Are hero is " + derekHero.name + "<br />");

// This won't change the name to Paul

var paulHero = new Hero("Paul");

document.write("Are hero is " + paulHero.name + "<br />");

// ---------- FACTORY PATTERN ----------

// The factory pattern can be used to generate different

// objects on request

function Sword(desc){

this.weaponType = "Sword";

this.metal = desc.metal || "Steel";

this.style = desc.style || "Longsword";

this.hasMagic = desc.hasMagic || false;

}

function Bow(desc){

this.weaponType = "Bow";

this.material = desc.material || "Wood";

this.style = desc.style || "Longbow";

this.hasMagic = desc.hasMagic || false;

}

function WeaponFactory(){};

WeaponFactory.prototype.makeWeapon = function(desc){

var weaponClass = null;

if(desc.weaponType === "Sword"){

weaponClass = Sword;

} else if (desc.weaponType === "Bow"){

weaponClass = Bow;

} else {

return false;

}

return new weaponClass(desc);

}

var myWeaponFact = new WeaponFactory();

var bladeFist = myWeaponFact.makeWeapon({

weaponType: "Sword",

metal: "Dark Iron",

style: "Scythe",

hasMagic: true

});

document.write(bladeFist.weaponType + " of type " + bladeFist.style + " crafted from " + bladeFist.metal + "<br />");

// ---------- DECORATOR PATTERN ----------

// The decorator pattern allows you alter an object at run time

function Pizza(price){

this.price = price || 10;

}

Pizza.prototype.getPrice = function(){

return this.price;

}

function ExtraCheese(pizza){

var prevPrice = pizza.price;

pizza.price = prevPrice + 1;

}

var myPizza = new Pizza(10);

ExtraCheese(myPizza);

document.write("Cost of Pizza : $" + myPizza.price + "<br />");

// ---------- OBSERVER PATTERN ----------

// A single object notifies many objects (observers) when a

// state change occurs

var Observable = function() {

this.subscribers = [];

}

Observable.prototype = {

subscribe: function(subscriber) {

// Add the subscriber object to the list

this.subscribers.push(subscriber);

},

unsubscribe: function(unsubscriber) {

// Cycle through the subscriber array and delete

// the unsubscriber

for (i = 0; i < this.subscribers.length; i++) {

if (this.subscribers[i] === unsubscriber) {

this.subscribers.splice(i, 1);

// We assume it only subscribed once so we

// leave after it is found

return unsubscriber.name;

}

}

},

publish: function(data) {

// Cycle through all subscribers and send them the update

for (i = 0; i < this.subscribers.length; i++) {

this.subscribers[i].receiveData(data);

}

}

};

var OrganFanny = {

name: "Organ Fanny",

receiveData: function(data){

document.write(this.name + " received your info : " + data + "<br />");

}

}

var BoldmanYaks = {

name: "Boldman Yaks",

receiveData: function(data){

document.write(this.name + " received your info : " + data + "<br />");

}

}

// Add subscribers and alert them

observable = new Observable();

observable.subscribe(OrganFanny);

observable.subscribe(BoldmanYaks);

observable.publish('IBM at $145.30');

document.write(observable.unsubscribe(OrganFanny) + " Unsubscribed<br />");

observable.publish('IBM at $145.33')

</script>

</body>

</html>

**ES6:**

/ ----- Method Notation in Object Property Definitions -----

// EC5 WAY

var addStuff = {

sum: function(num1, num2){

return num1 + num2;

}

};

document.write("1 + 2 = ", addStuff.sum(1,2) + "<br />");

// EC6 WAY

var addStuff = {

sum(num1, num2){

return num1 + num2;

}

}

document.write("1 + 2 = ", addStuff.sum(1,2) + "<br />");

// ----- CLASSES IN JAVASCRIPT -----

// EC5 WAY

var Point = function(xPos, yPos){

this.xPos = xPos;

this.yPos = yPos;

};

Point.prototype.getPos = function(){

return "X: " + this.xPos + " Y: " + this.yPos;

};

var point = new Point(100, 200);

document.write("Point Pos : " + point.getPos() + "<br />");

// EC6 WAY

class Point {

constructor(xPos, yPos){

this.xPos = xPos;

this.yPos = yPos;

}

getPos(){

return "X: " + this.xPos + " Y: " + this.yPos;

}

}

var point = new Point(100,200);

document.write("Point Pos : " + point.getPos() + "<br />");

// ----- MORE OOP IN JAVASCRIPT -----

// EC6 WAY

class Animal {

constructor(name){

this.name = name;

}

toString(){

return "Animal is named " + this.name;

}

// We can create static functions as well

static getAnimal(){

return new Animal("No Name");

}

}

class Dog extends Animal{

constructor(name, owner){

// We can call the super class now

super(name);

this.owner = owner;

}

toString(){

// You can call super class methods as well

return super.toString() + "<br />Dog is named " + this.name;

}

}

var rover = new Dog("Rover", "Paul");

document.write(rover.name + " is owned by " + rover.owner + "<br />");

document.write(rover.toString() + "<br />");

// Call the static function

var bowser = Animal.getAnimal();

document.write("Bowser info : " + bowser.toString() + "<br />");