**Intro to the Object Oriented Programming Challenge**

1. Create a Basic JavaScript Object

let dog = {

name: "Doggie",

numLegs: 4

};

2. Use Dot Notation to Access the Properties of an Object

let dog = {

name: "Spot",

numLegs: 4

};

// Add your code below this line

console.log(dog.name);

console.log(dog.numLegs);

3. Create a Method on an Object

let dog = {

name: "Spot",

numLegs: 4,

sayLegs: function() {return "This dog has " + dog.numLegs + " legs.";}

};

dog.sayLegs();

4. Make Code More Reusable with the “this” Keyword

let dog = {

name: "Spot",

numLegs: 4,

sayLegs: function() {return "This dog has " + this.numLegs + " legs.";}

};

dog.sayLegs();

5. Define a Constructor Function

function Dog() {

this.name = "Harry";

this.color = "White";

this.numLegs = 4;

}

6. Use a Constructor to Create Objects

function Dog() {

this.name = "Rupert";

this.color = "brown";

this.numLegs = 4;

}

// Add your code below this line

let hound = new Dog();

7. Extend Constructors to Receive Arguments

function Dog(name, color) {

this.name = name;

this.color = color;

this.numLegs = 4;

}

let terrier = new Dog("Snoopy", "white");

**8. Verify an Object’s Constructor with “instanceof”**

/\* jshint expr: true \*/

function House(numBedrooms) {

this.numBedrooms = numBedrooms;

}

// Add your code below this line

let myHouse = new House(3);

myHouse instanceof House;

**9. Understand Own Properties**

Name and numLegs are called own properties, because they are defined directly on the instance object. That means that duck and canary each has its own separate copy of these properties.

function Bird(name) {

this.name = name;

this.numLegs = 2;

}

let canary = new Bird("Tweety");

let ownProps = [];

// Add your code below this line

for (let property in canary) {

if(canary.hasOwnProperty(property)) {

ownProps.push(property);

}

}

console.log(ownProps);

**10. Use Prototype Properties to Reduce Duplicate Code**

function Dog(name) {

this.name = name;

}

Dog.prototype.numLegs = 4;

// Add your code above this line

let beagle = new Dog("Snoopy");

**11. Iterate Over All Properties**

You have now seen two kinds of properties: ownproperties and prototypeproperties. Ownproperties are defined directly on the object instance itself. And prototypeproperties are defined on the prototype.

function Dog(name) {

this.name = name;

}

Dog.prototype.numLegs = 4;

let beagle = new Dog("Snoopy");

let ownProps = [];

let prototypeProps = [];

// Add your code below this line

for(let property in beagle) {

if(beagle.hasOwnProperty(property)){

ownProps.push(property);

} else {

prototypeProps.push(property);

}

}

console.log(ownProps);

console.log(prototypeProps);

**12. Understand the Constructor Property**

function Dog(name) {

this.name = name;

}

// Add your code below this line

function joinDogFraternity(candidate) {

if(candidate.constructor === Dog) {

return true;

} else {

return false;

}

}

**13. Change the Prototype to a New Object**

function Dog(name) {

this.name = name;

}

Dog.prototype = {

// Add your code below this line

numLegs: 4,

eat: function() {

console.log("nom nom nom");

},

describe: function() {

console.log("My name is " + this.name);

}

};

**14. Remember to Set the Constructor Property when Changing the Prototype**

function Dog(name) {

this.name = name;

}

// Modify the code below this line

Dog.prototype = {

constructor: Dog,

numLegs: 2,

eat: function() {

console.log("nom nom nom");

},

describe: function() {

console.log("My name is " + this.name);

}

};

**15. Understand Where an Object’s Prototype Comes From**

function Dog(name) {

this.name = name;

}

let beagle = new Dog("Snoopy");

// Add your code below this line

Dog.prototype.isPrototypeOf(beagle);

\*\*\*checks the prototype of beagle

16. Understand the Prototype Chain

function Dog(name) {

this.name = name;

}

let beagle = new Dog("Snoopy");

Dog.prototype.isPrototypeOf(beagle); // => true

// Fix the code below so that it evaluates to true

Object.prototype.isPrototypeOf(Dog.prototype);

**17. Use Inheritance So You Don’t Repeat Yourself (DRY)**

function Cat(name) {

this.name = name;

}

Cat.prototype = {

constructor: Cat,

};

function Bear(name) {

this.name = name;

}

Bear.prototype = {

constructor: Bear,

};

function Animal() { }

Animal.prototype = {

constructor: Animal,

eat: function() {

console.log("nom nom nom");

}

};

18. Inherit Behaviors from a Supertype

function Animal() { }

Animal.prototype = {

constructor: Animal,

eat: function() {

console.log("nom nom nom");

}

};

// Add your code below this line

let duck = Object.create(Animal.prototype); // Change this line

let beagle = Object.create(Animal.prototype); // Change this line

duck.eat(); // Should print "nom nom nom"

beagle.eat(); // Should print "nom nom nom"

19. Set the Child’s Prototype to an Instance of the Parent

function Animal() { }

Animal.prototype = {

constructor: Animal,

eat: function() {

console.log("nom nom nom");

}

};

function Dog() { }

// Add your code below this line

Dog.prototype = Object.create(Animal.prototype);

let beagle = new Dog();

beagle.eat(); // Should print "nom nom nom"

20. Reset an Inherited Constructor Property

function Animal() { }

function Bird() { }

function Dog() { }

Bird.prototype = Object.create(Animal.prototype);

Dog.prototype = Object.create(Animal.prototype);

// Add your code below this line

Bird.prototype.constructor = Bird;

Dog.prototype.constructor = Dog;

let duck = new Bird();

let beagle = new Dog();

21. Add Methods After Inheritance

function Animal() { }

Animal.prototype.eat = function() { console.log("nom nom nom"); };

function Dog() { }

// Add your code below this line

Dog.prototype = Object.create(Animal.prototype);

Dog.prototype.constructor = Dog;

Dog.prototype.bark = function() {

console.log("Woof!");

}

// Add your code above this line

let beagle = new Dog();

beagle.eat(); // Should print "nom nom nom"

beagle.bark(); // Should print "Woof!"

22. Override Inherited Methods

function Bird() { }

Bird.prototype.fly = function() { return "I am flying!"; };

function Penguin() { }

Penguin.prototype = Object.create(Bird.prototype);

Penguin.prototype.constructor = Penguin;

// Add your code below this line

Penguin.prototype.fly = function() {

return "Alas, this is a flightless bird.";

}

// Add your code above this line

let penguin = new Penguin();

console.log(penguin.fly());

23. Use a Mixin to Add Common Behavior Between Unrelated Objects

For unrelated objects, it's better to use mixins. A mixinallows other objects to use a collection of functions.

let bird = {

name: "Donald",

numLegs: 2

};

let boat = {

name: "Warrior",

type: "race-boat"

};

// Add your code below this line

let glideMixin = function(obj) {

obj.glide = function() {

console.log("Gliding!");

}

}

glideMixin(bird);

glideMixin(boat);

bird.glide();

boat.glide();

**24. Use Closure to Protect Properties Within an Object from Being Modified Externally**

In JavaScript, a function always has access to the context in which it was created. This is called closure.

function Bird() {

let weight = 15;

this.getWeight = function() {

return weight;

}

}

25. Understand the Immediately Invoked Function Expression (IIFE)

A common pattern in JavaScript is to execute a function as soon as it is declared.

Original Function:

function makeNest() {

console.log("A cozy nest is ready");

}

makeNest();

IIFE:

(function () {

console.log("A cozy nest is ready");

})();

Note that the function has no name and is not stored in a variable. The two parentheses () at the end of the function expression cause it to be immediately executed or invoked. This pattern is known as an immediately invoked function expressionor IIFE.

**26. Use an IIFE to Create a Module**

An immediately invoked function expression(IIFE) is often used to group related functionality into a single object or module.

The advantage of the module pattern is that all of the motion behaviors can be packaged into a single object that can then be used by other parts of your code.

let isCuteMixin = function(obj) {

obj.isCute = function() {

return true;

};

};

let singMixin = function(obj) {

obj.sing = function() {

console.log("Singing to an awesome tune");

};

};

To:

let funModule = (function () {

return {

isCuteMixin: function(obj) {

obj.isCute = function() {

return true;

};

},

singMixin: function() {

obj.sing = function() {

console.log("Singing to an awesome tune");

};

}

}

}) ();

\*This IIFE returns an object “funModule”. This returned object contains all of the mixin behaviors as properties of the object.