Programación orientada a objetos:

1.- Crea un objeto básico en JavaScript:

let dog = {
  name : "Rata",
  numLegs: 4
};
	Crea un objeto dog con las propiedades name y numLegs y asignales una cadena y un número respectivamente.

2.- Utiliza notación de puntos para acceder a las propiedades de un objeto:

let dog = {
  name: "Spot",
  numLegs: 4
};
console.log(dog.name);
console.log(dog.numLegs);
	Imprime ambas propiedades del objeto en tu consola, utilizando la notación de puntos.

3.- Crea un método en un objeto:

let dog = {
  name: "Spot",
  numLegs: 4,
  sayLegs: function(){
  return "This dog has "
  + dog.numLegs + " legs."}
};
dog.sayLegs();
	Usando el objeto dog, asignale un método llamado sayLegs. El método debe devolver la frase This dog has 4 legs.

4.- Haz el código más reutilizable con la palabra clave “this”:

let dog = {
  name: "Spot",
  numLegs: 4,
  sayLegs: function() {
  return "This dog has "
  + this.numLegs + " legs.";}
};
dog.sayLegs();
	Modifica el método dog.sayLegs para eliminar cualquier referencia a dog.

5.- Define una función “Constructor”:

Las funciones constructoras crean nuevos objetos y definen las propiedades y comportamientos que pertenecen al objeto. Están definidos con un nombre en mayúsculas, para distinguirlos de las funciones no constructoras. Utilizan la palabra clave this para establecer propiedades del objeto que crearán. Los constructores definen propiedades y comportamientos en vez de devolverlos como un valor como lo harían otras funciones.

function Dog(){
  this.name = "Rata";
  this.color = "white and brown";
  this.numLegs = 4;
}
	Crea un constructor, Dog, con las propiedades name, color y numLegs que se establecen a una cadena, una cadena y un número, respectivamente.

6.- Utiliza un constructor para crear objetos:

let hound = new Dog();	Utiliza el constructor Dog de la última lección para crear una nueva instancia de Dog, asignandolo a una variable hound.

7.- Extender constructores para recibir argumentos:

function Dog(name, color) {
  this.name = name;
  this.color = color;
  this.numLegs = 4;
}
let terrier = new Dog("Bruce","back");
	Crea otro constructor Dog. Esta vez, configurarlo para que tome los parámetros name y color, y ten la propiedad numLegs fija a 4. Crea un nuevo objeto almacenado en la variable terrier. Pasale dos cadenas de texto como argumento.

8.- Verifica el constructor de un objeto con “instanceof”:

function House(numBedrooms) {
  this.numBedrooms = numBedrooms;
}
let myHouse = new House(2);
myHouse instanceof House;
	Crea una nueva instancia del constructor House, llamándola myHouse y pasando el número de habitaciones. Luego, usa instanceof para verificar que es una instancia de House.

9.- Comprender las propiedades directas:

function Bird(name) {
  this.name = name;
  this.numLegs = 2;
}
 	
let canary = new Bird("Tweety");
let ownProps = [];
 	
for (let property in canary){
  if(canary.hasOwnProperty(property)){
  ownProps.push(property);
  }
}
	Agrega todas las propiedades directas de canary al arreglo ownProps.

10.- Utiliza propiedades “prototype” para reducir código duplicado:

Las propiedades con prototype se comparten entre todas las instancias del constructor.

function Dog(name) {
  this.name = name;
}
Dog.prototype.numLegs = 4;
let beagle = new Dog("Snoopy");
	Añade una propiedad numLegs al prototype de Dog.

11.- Itera sobre todas las propiedades:

Hay dos tipos de propiedades: propiedades directas y propiedades prototype. Las directas se definen directamente en la propiedad instancia del objeto. Y las prototype se definen en el prototype.

function Dog(name) {
  this.name = name;
}
Dog.prototype.numLegs = 4;
let beagle = new Dog("Snoopy");
 	
let ownProps = [];
let prototypeProps = [];
 	
for (let property in beagle){
  if(beagle.hasOwnProperty(property)){
  ownProps.push(property);
  } else {
  prototypeProps.push(property);
  }
}
	Agrega todas las propiedades directas de beagle al arreglo ownProps. Agrega todas las propiedades prototype de Dog al arreglo prototypeProps.

12.- Entiende la propiedad constructor:

La propiedad constructor hace referencia a la función constructora que creó la instancia. La ventaja de la propiedad constructor es que es posible verificar esta propiedad para averiguar qué tipo de objeto es. Dado que la propiedad constructor se puede reescribir, es mejor utilizar el método instanceof para verificar el tipo de un objeto.

function joinDogFraternity(candidate) {
  if(candidate.constructor === Dog){
  return true;
  } else {
  return false;
  }
}
	Escribe una función joinDogFraternity que tome como parámetro candidate y que con la propiedad constructor devuelva true si el candidato es un Dog y false si no lo es.

13.- Cambia el prototype a un nuevo objeto:

function Dog(name) {
  this.name = name;
}
Dog.prototype = {
  numLegs: 4,
  eat: function(){
  console.log("yom yom yom");
  },
  describe: function(){
  console.log("My name is " + this.name);
  }
};
	Agrega la propiedad numLegs, y los métodos eat() y describe() al prototype de Dog, estableciendo prototype a un nuevo objeto.

14.- Recuerda establecer la propiedad “constructor” al cambiar el prototype:

Un efecto secundario de ajustar manualmente el prototype a un nuevo objeto es que elimina la propiedad constructor. Esta propiedad puede ser usada para verificar cuál función creó la instancia. Sin embargo, dado que la propiedad ha sido sobreescrita, ahora devuelve resultados falsos.
Para solucionar esto, cada vez que un prototype se establece de forma manual para un nuevo objeto, recuerda definir la propiedad constructor.

constructor: Dog,	Define la propiedad constructor en el Dog prototype.

15.- Entendiendo de dónde viene el prototype de un objeto:

Los objetos heredan su prototype del constructor que los creó. Puedes mostrar esta relación con el método isPrototypeOf().

function Dog(name) {
  this.name = name;
}
let beagle = new Dog("Snoopy");
Dog.prototype.isPrototypeOf(beagle);
	Utiliza isPrototypeOf para comprobar el prototype de beagle.

16.- Comprende la cadena “prototype”:

Todos los objetos en JavaScript tienen prototype (con algunas excepciones). Además, el prototype de un objeto, es otro objeto en sí mismo.

Object.prototype.isPrototypeOf (Dog.prototype);	Modifica el código para mostrar la cadena de prototypes correcta.

17.- Usa herencia para que no te repitas:

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");
  }
};
	El método eat se repite tanto en Cat como en Bear. Edita el código utilizando el principio DRY, moviendo el método eat al supertype Animal.

18.- Hereda comportamientos de un supertipo (supertype):

Object.create(obj) creá un objeto nuevo y establece obj como el prototype del nuevo objeto. El prototype es como la “receta” para crear un objeto.

function Animal() { }
 	
Animal.prototype = {
  constructor: Animal,
  eat: function() {
  console.log("nom nom nom");
  }
};
 	
let duck = Object.create(Animal.prototype);
let beagle
= Object.create(Animal.prototype);
	Utiliza Object.create() para crear dos instancias de Animal llamadas duck y beagle.

19.- Establece el prototype de un hijo (subtype) para una instancia de un padre (supertype):

function Animal() { }
 	
Animal.prototype = {
  constructor: Animal,
  eat: function() {
  console.log("nom nom nom");
  }
};
 	
function Dog() { }
 	
Dog.prototype =
Object.create(Animal.prototype);
 	
let beagle = new Dog();
	Modifica el código para que las instancias de Dog hereden de Animal.

20.- Restablece una propiedad “constructor” heredada:

Cuando un objeto hereda el prototype de otro objeto, también hereda la propiedad del constructor del supertype. Por ejemplo:

```
function Bird() { }
Bird.prototype = Object.create(Animal.prototype);
let duck = new Bird();
duck.constructor
```

Pero duck y todas las instancias de Bird deberían mostrar que fueron construidas por Bird y no Animal. Para ello, puedes establecer manualmente la propiedad del constructor Bird al objeto Bird.

```
Bird.prototype.constructor = Bird;
duck.constructor
```

function Animal() { }
function Bird() { }
function Dog() { }
 	
Bird.prototype =
Object.create(Animal.prototype);
Dog.prototype =
Object.create(Animal.prototype);
 	
Bird.prototype.constructor = Bird;
Dog.prototype.constructor = Dog;
 	
let duck = new Bird();
let beagle = new Dog();
	Corrige el código para que duck.constructor y beagle.constructor devuelvan sus constructores respectivos.

21.- Añade métodos después de la herencia:

Una función constructor que hereda su objeto prototype de una función constructor padre (supertype) puede seguir teniendo sus propios métodos además de los heredados.

function Animal() { }
Animal.prototype.eat = function() {
console.log("nom nom nom"); };
 	
function Dog() { }
Dog.prototype =
Object.create(Animal.prototype);
Dog.prototype.constructor = Dog;
Dog.prototype.bark = function(){
  console.log("Woof!")
}
 	
let beagle = new Dog();
	Añade el código necesario para que el objeto Dog herede de Animal y el constructor prototype de Dog sea establecido en Dog. A continuación agrega el método bark() al objeto Dog, para que beagle contenga bark() y eat(). El método bark() debe imprimir Woof! en consola.

22.- Sobreescribir métodos heredados:

function Bird() { }
 	
Bird.prototype.fly = function() { return "I am flying!"; };
 	
function Penguin() { }
Penguin.prototype =
Object.create(Bird.prototype);
Penguin.prototype.constructor = Penguin;
 	
Penguin.prototype.fly = function(){
  return "Alas, this is a flightless bird."
}
 	
let penguin = new Penguin();
console.log(penguin.fly());
	Sobreescribe el método fly() para Penguin, de manera que devuelva la cadena de texto “Alas, this is a flightless bird.”.

23.- Utiliza un “mixin” para añadir un comportamiento común entre objetos no relacionados:

La herencia no funciona bien con objetos que no están relacionados. Para estos casos es mejor utilizar mixins. Un “mixin” permite a otros objetos utilizar una colección de funciones. La siguiente función toma cualquier objeto y le da el método fly.

```
let flyMixin = function(obj) {
  obj.fly = function() {
    console.log("Flying, wooosh!");
  }
};
```

```
flyMixin(bird);
flyMixin(plane);
```

let bird = {
  name: "Donald",
  numLegs: 2
};
 	
let boat = {
  name: "Warrior",
  type: "race-boat"
};
 	
let glideMixin = function(obj){
  obj.glide = function(){
  cosole.log("Can glide.");
  }
};
 	
glideMixin(bird);
glideMixin(boat);
	Crea un mixin llamado glideMixin que defina un método llamado glide. Luego utiliza el glideMixin para dar a bird y boat la habilidad de planear.

24.- Utiliza closures para evitar que las propiedades de un objeto se puedan modificar desde fuera:

En el desafío anterior bird tenía una propiedad pública name. Se considera pública porque se puede acceder y cambiar fuera de la definición de bird. Por tanto cualquier parte de tu código puede cambiar fácilmente el nombre “name” de bird a cualquier valor.Piensa en cosas como contraseñas o cuentas bancarias que se pueden cambiar fácilmente desde cualquier parte de tu código. Esto podría generar muchos problemas.
La forma más sencilla de hacer privada está propiedad pública sería creando una variable dentro de la función constructora. Esto cambia el alcance de está variable para que este disponible solo desde la función constructora en lugar de tener un alcance global (global scope).

function Bird() {
  let weight = 15;
  this.getWeight = function(){
  return weight;
  }
}
	Cambia como weight es declarada en la función Bird para que sea una variable privada. Después, crea un método getWeight que devuelva el valor 15 para weight.

25.- Comprende las funciones que son invocadas inmediatamente (IIFE):

Un patrón muy común en JavaScript es la ejecución de una función apenas declarada. Está es una expresión de función anónima que se ejecuta de inmediato. Los paréntesis al final de la función hace que se ejecute inmediatamente.

(function () {
  console.log("Chirp, chirp!");
})();
	Ten en cuenta que la función no tiene nombre y no se almacena en un valor.

(function () { console.log("A cozy nest is ready"); })();	Reescribe la función makeNest y elimina su llamada, para que sea una expresión de función anónima inmediatamente invocada (IIFE).

26.- Utiliza una IIFE para crear un módulo:

Una expresión de función inmediatamente invocada (IIFE) se utiliza a menudo para agrupar la funcionalidad relacionada en un solo objeto o módulo. Por ejemplo, en un desafío anterior se definieron dos mixins que podemos agrupar en un módulo de la siguiente forma:

```
function glideMixin(obj) {
  obj.glide = function() {
    console.log("Gliding on the water");
  };
}
function flyMixin(obj) {
  obj.fly = function() {
    console.log("Flying, wooosh!");
  };
}
```

```
let motionModule = (function () {
  return {
    glideMixin: function(obj) {
      obj.glide = function() {
        console.log("Gliding on the water");
      };
    },
    flyMixin: function(obj) {
      obj.fly = function() {
        console.log("Flying, wooosh!");
      };
    }
  }
})();
```

Ten en cuenta que has invocado una IIFE que devuelve un objeto motionModule. EL objeto devuelto contiene todos los comportamientos de los mixins como propiedades del objeto. La ventaja de este patrón del módulo es que todos los comportamientos de movimiento pueden ser empaquetados en un solo objeto que puede ser usado por otras partes del código, tal que así:

```
motionModule.glideMixin(duck);
duck.glide();
```

let funModule = (function (){
  return {
  isCuteMixin: function(obj){
  obj.isCute = function(){
  return true;
  };
  },
  singMixin: function(obj){
  obj.sing = function(){
  console.log("Singing to an awesome tune");
  };
  } } })();
	Crea un módulo llamado funModule para envolver los dos mixins isCuteMixin y singMixin. funModule debe devolver un objeto.