**Programación orientada a objetos**

**Crear un objeto JavaScript básico**

Piense en cosas que la gente ve todos los días, como automóviles, tiendas y pájaros. Todos estos son objetos : cosas tangibles que las personas pueden observar e interactuar.

¿Cuáles son algunas cualidades de estos objetos? Un coche tiene ruedas. Las tiendas venden artículos. Los pájaros tienen alas.

Estas cualidades, o propiedades , definen lo que constituye un objeto. Tenga en cuenta que los objetos similares comparten las mismas propiedades, pero pueden tener diferentes valores para esas propiedades. Por ejemplo, todos los autos tienen ruedas, pero no todos los autos tienen la misma cantidad de ruedas.

Los objetos en JavaScript se usan para modelar objetos del mundo real, dándoles propiedades y comportamientos como sus contrapartes del mundo real. Aquí hay un ejemplo usando estos conceptos para crear un duckobjeto:

let duck = {

name: "Aflac",

numLegs: 2

};

Este duckobjeto tiene dos pares propiedad/valor: a namede Aflacy a numLegsde 2.

**let dog = {**

**name: "rocky" ,**

**numLegs: 4**

**};**

**Use la notación de puntos para acceder a las propiedades de un objeto**

El último desafío creó un objeto con varias propiedades. Ahora verá cómo acceder a los valores de esas propiedades. Aquí hay un ejemplo:

**let duck = {**

**name: "Aflac",**

**numLegs: 2**

**};**

**console.log(duck.name);**

La notación de puntos se usa en el nombre del objeto, duckseguido del nombre de la propiedad, namepara acceder al valor de Aflac.

**let dog = {**

**name: "Spot",**

**numLegs: 4**

**};**

**// Only change code below this line**

**console.log(dog.name)**

**console.log(dog.numLegs)**

**Crear un método en un objeto**

Los objetos pueden tener un tipo especial de propiedad, llamado método .

Los métodos son propiedades que son funciones. Esto agrega un comportamiento diferente a un objeto. Aquí está el duckejemplo con un método:

**let duck = {**

**name: "Aflac",**

**numLegs: 2,**

**sayName: function() {return "The name of this duck is " + duck.name + ".";}**

**};**

**duck.sayName();**

El ejemplo agrega el sayNamemétodo, que es una función que devuelve una oración que da el nombre del duck. Observe que el método accedió a la namepropiedad en la declaración de devolución usando duck.name. El próximo desafío cubrirá otra forma de hacer esto.

**let dog = {**

**name: "Spot",**

**numLegs: 4,**

**sayLegs: function() {**

**return "This dog has " + dog.numLegs + " legs.";**

**}**

**};**

**dog.sayLegs();**

**Haga que el código sea más reutilizable con esta palabra clave**

El último desafío introdujo un método para el duckobjeto. Usó duck.namela notación de puntos para acceder al valor de la namepropiedad dentro de la declaración de devolución:

**sayName: function() {return "The name of this duck is " + duck.name + ".";}**

Si bien esta es una forma válida de acceder a la propiedad del objeto, aquí hay una trampa. Si el nombre de la variable cambia, cualquier código que haga referencia al nombre original también deberá actualizarse. En una definición de objeto corta, no es un problema, pero si un objeto tiene muchas referencias a sus propiedades, hay una mayor posibilidad de error.

**Una forma de evitar estos problemas es con la thispalabra clave:**

**let duck = {**

**name: "Aflac",**

**numLegs: 2,**

**sayName: function() {return "The name of this duck is " + this.name + ".";}**

**};**

thises un tema profundo, y el ejemplo anterior es solo una forma de usarlo. En el contexto actual, thisse refiere al objeto con el que está asociado el método: duck. Si el nombre del objeto se cambia a mallard, no es necesario encontrar todas las referencias ducken el código. Hace que el código sea reutilizable y más fácil de leer.

**let dog = {**

**name: "Spot",**

**numLegs: 4,**

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

**};**

**dog.sayLegs();**

**Definir una función constructora**

Los constructores son funciones que crean nuevos objetos. Definen propiedades y comportamientos que pertenecerán al nuevo objeto. Piense en ellos como un modelo para la creación de nuevos objetos.

Aquí hay un ejemplo de un constructor:

**function Bird() {**

**this.name = "Albert";**

**this.color = "blue";**

**this.numLegs = 2;**

**}**

Este constructor define un Birdobjeto con las propiedades name, colory numLegsestablecidas en Albert, blue y 2, respectivamente. Los constructores siguen algunas convenciones:

Los constructores se definen con un nombre en mayúsculas para distinguirlos de otras funciones que no lo son constructors.

Los constructores usan la palabra clave thispara establecer las propiedades del objeto que crearán. Dentro del constructor, thisse refiere al nuevo objeto que creará.

Los constructores definen propiedades y comportamientos en lugar de devolver un valor como lo harían otras funciones.

**function Dog() {**

**this.name = "rocky";**

**this.color = "black";**

**this.numLegs = 4;**

**}**

**Usar un constructor para crear objetos**

**Aquí está el Birdconstructor del desafío anterior:**

**function Bird() {**

**this.name = "Albert";**

**this.color = "blue";**

**this.numLegs = 2;**

**}**

**let blueBird = new Bird();**

NOTA: this dentro del constructor siempre se refiere al objeto que se está creando.

Tenga en cuenta que el newoperador se usa cuando se llama a un constructor. Esto le dice a JavaScript que cree una nueva instancia de Birdcalled blueBird. Sin el newoperador, thisdentro del constructor no apuntaría al objeto recién creado, dando resultados inesperados. Ahora blueBirdtiene todas las propiedades definidas dentro del Birdconstructor:

**blueBird.name;**

**blueBird.color;**

**blueBird.numLegs;**

Al igual que cualquier otro objeto, se puede acceder a sus propiedades y modificarlas:

**blueBird.name = 'Elvira';**

**blueBird.name;**

**function Dog() {**

**this.name = "Rupert";**

**this.color = "brown";**

**this.numLegs = 4;**

**}**

**// Only change code below this line**

**let hound = new Dog**

**Extender constructores para recibir argumentos**

Los constructores Birdy Dogdel último desafío funcionaron bien. Sin embargo, observe que todos Birdslos que se crean con el Birdconstructor se denominan automáticamente Albert, son de color azul y tienen dos patas. ¿Qué pasa si quieres pájaros con diferentes valores de nombre y color? Es posible cambiar las propiedades de cada ave manualmente pero eso sería mucho trabajo:

**let swan = new Bird();**

**swan.name = "Carlos";**

**swan.color = "white";**

Suponga que está escribiendo un programa para realizar un seguimiento de cientos o incluso miles de aves diferentes en un aviario. Llevaría mucho tiempo crear todos los pájaros y luego cambiar las propiedades a diferentes valores para cada uno. Para crear más fácilmente diferentes Birdobjetos, puede diseñar su constructor Bird para aceptar parámetros:

**function Bird(name, color) {**

**this.name = name;**

**this.color = color;**

**this.numLegs = 2;**

**}**

Luego pase los valores como argumentos para definir cada ave única en el Birdconstructor: let cardinal = new Bird("Bruce", "red");Esto da una nueva instancia de Birdcon namey colorpropiedades establecidas en Brucey red, respectivamente. La numLegspropiedad todavía está establecida en 2. cardinalTiene estas propiedades:

**cardinal.name**

**cardinal.color**

**cardinal.numLegs**

El constructor es más flexible. Ahora es posible definir las propiedades de cada uno Birden el momento en que se crea, que es una forma en que los constructores de JavaScript son tan útiles. Agrupan objetos en función de características y comportamientos compartidos y definen un modelo que automatiza su creación.

**function Dog(name, color) {**

**this.name = name;**

**this.color = color;**

**this.numLegs = 4;**

**}**

**let terrier = new Dog("rocky","dante");**

**Verificar el constructor de un objeto con instanceof**

Cada vez que una función constructora crea un nuevo objeto, se dice que ese objeto es una instancia de su constructor. JavaScript brinda una manera conveniente de verificar esto con el instanceofoperador. instanceofle permite comparar un objeto con un constructor, devolviendo trueo falseen función de si ese objeto se creó o no con el constructor. Aquí hay un ejemplo:

**let Bird = function(name, color) {**

**this.name = name;**

**this.color = color;**

**this.numLegs = 2;**

**}**

**let crow = new Bird("Alexis", "black");**

**crow instanceof Bird;**

Este instanceofmétodo devolvería true.

Si se crea un objeto sin usar un constructor, instanceofse verificará que no sea una instancia de ese constructor:

**let canary = {**

**name: "Mildred",**

**color: "Yellow",**

**numLegs: 2**

**};**

**canary instanceof Bird;**

Este instanceofmétodo devolvería false.

**function House(numBedrooms) {**

**this.numBedrooms = numBedrooms;**

**}**

**// Only change code below this line**

**let myHouse = new House (2);**

**myHouse instanceof House;**

**Comprender las propiedades propias**

En el siguiente ejemplo, el Birdconstructor define dos propiedades: namey numLegs:

**function Bird(name) {**

**this.name = name;**

**this.numLegs = 2;**

**}**

**let duck = new Bird("Donald");**

**let canary = new Bird("Tweety");**

namey numLegsse llaman propiedades propias , porque se definen directamente en el objeto de instancia. Eso significa que duckcada canaryuno tiene su propia copia separada de estas propiedades. De hecho, cada instancia de Birdtendrá su propia copia de estas propiedades. El siguiente código agrega todas las propiedades propias de duckla matriz ownProps:

**let ownProps = [];**

**for (let property in duck) {**

**if(duck.hasOwnProperty(property)) {**

**ownProps.push(property);**

**}**

**}**

**console.log(ownProps);**

La consola mostraría el valor ["name", "numLegs"].

**function Bird(name) {**

**this.name = name;**

**this.numLegs = 2;**

**}**

**let canary = new Bird("Tweety");**

**let ownProps = [];**

**// Only change code below this line**

**for (let property in canary){**

**if(canary.hasOwnProperty(property)){**

**ownProps.push(property)**

**}**

**}**

**Utilice las propiedades del prototipo para reducir el código duplicado**

Dado que numLegsprobablemente tendrá el mismo valor para todas las instancias de Bird, esencialmente tiene una variable duplicada numLegsdentro de cada Birdinstancia.

Esto puede no ser un problema cuando solo hay dos instancias, pero imagínese si hay millones de instancias. Eso sería un montón de variables duplicadas.

Una mejor manera es usar el prototypeof Bird. Las propiedades de prototypese comparten entre TODAS las instancias de Bird. Aquí se explica cómo agregar numLegsa Bird prototype:

**Bird.prototype.numLegs = 2;**

Ahora todas las instancias de Birdtienen la numLegspropiedad.

**console.log(duck.numLegs);**

**console.log(canary.numLegs);**

Dado que todas las instancias tienen automáticamente las propiedades en prototype, piense en a prototypecomo una "receta" para crear objetos. Tenga en cuenta que prototypefor ducky canaryes parte del Birdconstructor como Bird.prototype.

**function Dog(name) {**

**this.name = name;**

**}**

**Dog.prototype.numLegs = 4;**

**// Add your code above this line**

**let beagle = new Dog("Snoopy");**

**Iterar sobre todas las propiedades**

Ya has visto dos tipos de propiedades: propiedades propias y prototypepropiedades. Las propiedades propias se definen directamente en la propia instancia del objeto. Y las propiedades del prototipo se definen en el archivo prototype.

**function Bird(name) {**

**this.name = name; //own property**

**}**

**Bird.prototype.numLegs = 2; // prototype property**

**let duck = new Bird("Donald");**

Así es como agrega ducklas propiedades propias a la matriz ownPropsy prototypelas propiedades a la matriz prototypeProps:

**let ownProps = [];**

**let prototypeProps = [];**

**for (let property in duck) {**

**if(duck.hasOwnProperty(property)) {**

**ownProps.push(property);**

**} else {**

**prototypeProps.push(property);**

**}**

**}**

**console.log(ownProps);**

**console.log(prototypeProps);**

console.log(ownProps)se mostraría ["name"]en la consola y console.log(prototypeProps)se mostraría ["numLegs"].

**for (let property in beagle) {**

**if (Dog.hasOwnProperty(property)) {**

**ownProps.push(property);**

**} else {**

**prototypeProps.push(property);**

**}**

**}**

**Comprender la propiedad del constructor**

**Hay una constructorpropiedad especial ubicada en las instancias del objeto ducky beagleque se creó en los desafíos anteriores:**

**let duck = new Bird();**

**let beagle = new Dog();**

**console.log(duck.constructor === Bird);**

**console.log(beagle.constructor === Dog);**

**Ambas console.logllamadas se mostrarían trueen la consola.**

Tenga en cuenta que la constructorpropiedad es una referencia a la función constructora que creó la instancia. La ventaja de la constructorpropiedad es que es posible verificar esta propiedad para averiguar qué tipo de objeto es. Aquí hay un ejemplo de cómo se podría usar esto:

**function joinBirdFraternity(candidate) {**

**if (candidate.constructor === Bird) {**

**return true;**

**} else {**

**return false;**

**}**

**}**

Nota: dado que la constructorpropiedad se puede sobrescribir (lo cual se tratará en los siguientes dos desafíos), generalmente es mejor usar el instanceofmétodo para verificar el tipo de un objeto.

**function Dog(name) {**

**this.name = name;**

**}**

**// Only change code below this line**

**function joinDogFraternity(candidate) {**

**return candidate.constructor === Dog;**

**}**

**Cambiar el prototipo a un nuevo objeto**

Hasta ahora ha estado agregando propiedades a los prototypeindividualmente:

**Bird.prototype.numLegs = 2;**

Esto se vuelve tedioso después de más de unas pocas propiedades.

**Bird.prototype.eat = function() {**

**console.log("nom nom nom");**

**}**

**Bird.prototype.describe = function() {**

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

**}**

Una forma más eficiente es establecer el prototypeen un nuevo objeto que ya contiene las propiedades. De esta manera, las propiedades se agregan todas a la vez:

**Bird.prototype = {**

**numLegs: 2,**

**eat: function() {**

**console.log("nom nom nom");**

**},**

**describe: function() {**

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

**}**

**};**

**function Dog(name) {**

**this.name = name;**

**}**

**Dog.prototype = {**

**// Only change code below this line**

**numLegs: 2,**

**eat: function() {**

**console.log("nom nom nom");**

**},**

**describe: function() {**

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

**}**

**};**

**Recuerde establecer la propiedad del constructor**

**al cambiar el prototipo**

Hay un efecto secundario crucial de configurar manualmente el prototipo en un nuevo objeto. ¡ Borra la constructorpropiedad! Esta propiedad se puede usar para verificar qué función constructora creó la instancia, pero dado que la propiedad se sobrescribió, ahora da resultados falsos:

**duck.constructor === Bird;**

**duck.constructor === Object;**

**duck instanceof Bird;**

En orden, estas expresiones se evaluarían como false, truey true.

Para solucionar esto, siempre que un prototipo se establezca manualmente en un nuevo objeto, recuerde definir la constructorpropiedad:

**Bird.prototype = {**

**constructor: Bird,**

**numLegs: 2,**

**eat: function() {**

**console.log("nom nom nom");**

**},**

**describe: function() {**

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

**}**

**};**

**function Dog(name) {**

**this.name = name;**

**}**

**// Only change code below this line**

**Dog.prototype = {**

**constructor: Dog,**

**numLegs: 4,**

**eat: function() {**

**console.log("nom nom nom");**

**},**

**describe: function() {**

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

**}**

**};**

**Entender de dónde viene el prototipo de un objeto**

Al igual que las personas heredan genes de sus padres, un objeto hereda prototypedirectamente de la función constructora que lo creó. Por ejemplo, aquí el Birdconstructor crea el duckobjeto:

**function Bird(name) {**

**this.name = name;**

**}**

**let duck = new Bird("Donald");**

duckhereda prototypede la Birdfunción constructora. Puedes mostrar esta relación con el isPrototypeOfmétodo:

**Bird.prototype.isPrototypeOf(duck);**

Esto volvería true.

**function Dog(name) {**

**this.name = name;**

**}**

**let beagle = new Dog("Snoopy");**

**// Only change code below this line**

**Dog.prototype.isPrototypeOf(beagle);**

**Comprender la cadena de prototipos**

Todos los objetos en JavaScript (con algunas excepciones) tienen una extensión prototype. Además, un objeto prototypeen sí mismo es un objeto.

**function Bird(name) {**

**this.name = name;**

**}**

**typeof Bird.prototype;**

¡ Porque a prototypees un objeto, a prototypepuede tener el suyo propio prototype! En este caso, el prototypede Bird.prototypees Object.prototype:

**Object.prototype.isPrototypeOf(Bird.prototype);**

¿Cómo es esto útil? Puede recordar el hasOwnPropertymétodo de un desafío anterior:

**let duck = new Bird("Donald");**

**duck.hasOwnProperty("name");**

El hasOwnPropertymétodo se define en Object.prototype, al que puede acceder Bird.prototype, al que luego puede acceder duck. Este es un ejemplo de la prototypecadena. En esta prototypecadena, Birdes el supertypefor duck, mientras que duckes el subtype. Objectes a supertypepara ambos Birdy duck. Objectes un supertypepara todos los objetos en JavaScript. Por lo tanto, cualquier objeto puede usar el hasOwnPropertymétodo

**function Dog(name) {**

**this.name = name;**

**}**

**let beagle = new Dog("Snoopy");**

**Dog.prototype.isPrototypeOf(beagle); // yields true**

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

**Object.prototype.isPrototypeOf(Dog.prototype);**

**Usa la herencia para no repetirte**

Hay un principio en la programación llamado Don't Repeat Yourself (DRY) . La razón por la que el código repetido es un problema es porque cualquier cambio requiere corregir el código en varios lugares. Esto generalmente significa más trabajo para los programadores y más espacio para errores.

Observe en el siguiente ejemplo que el describemétodo es compartido por Birdy Dog:

**Bird.prototype = {**

**constructor: Bird,**

**describe: function() {**

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

**}**

**};**

**Dog.prototype = {**

**constructor: Dog,**

**describe: function() {**

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

**}**

**};**

El describemétodo se repite en dos lugares. El código se puede editar para seguir el principio DRY creando un supertype(o padre) llamado Animal:

**function Animal() { };**

**Animal.prototype = {**

**constructor: Animal,**

**describe: function() {**

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

**}**

**};**

Dado que Animalincluye el describemétodo, puede eliminarlo de Birdy Dog:

**Bird.prototype = {**

**constructor: Bird**

**};**

**Dog.prototype = {**

**constructor: Dog**

**};**

**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");**

**}**

**};**

**Heredar comportamientos de un supertipo**

En el desafío anterior, creaste un supertypellamado Animalque definía los comportamientos compartidos por todos los animales:

**function Animal() { }**

**Animal.prototype.eat = function() {**

**console.log("nom nom nom");**

**};**

Este y el próximo desafío cubrirán cómo reutilizar los métodos de Animaladentro Birdy Dogsin definirlos nuevamente. Utiliza una técnica llamada herencia. Este desafío cubre el primer paso: crear una instancia del supertype(o padre). Ya conoce una forma de crear una instancia de Animaluso del newoperador:

**let animal = new Animal();**

Hay algunas desventajas al usar esta sintaxis para la herencia, que son demasiado complejas para el alcance de este desafío. En cambio, aquí hay un enfoque alternativo sin esas desventajas:

**let animal = Object.create(Animal.prototype);**

Object.create(obj)crea un nuevo objeto y lo establece objcomo el nuevo objeto prototype. Recuerde que prototypees como la "receta" para crear un objeto. Al configurar el prototypede animalpara que sea el prototypede Animal, le está dando a la animalinstancia la misma "receta" que cualquier otra instancia de Animal.

**animal.eat();**

**animal instanceof Animal;**

El instanceofmétodo aquí devolvería true.

**function Animal() { }**

**Animal.prototype = {**

**constructor: Animal,**

**eat: function() {**

**console.log("nom nom nom");**

**}**

**};**

**// Only change code below this line**

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

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

**Establecer el prototipo del**

**niño en una instancia del padre**

En el desafío anterior, vio el primer paso para heredar el comportamiento del supertipo (o padre) Animal: crear una nueva instancia de Animal.

Este desafío cubre el siguiente paso: configurar el prototypedel subtipo (o secundario), en este caso, Birdpara que sea una instancia de Animal.

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

Recuerda que prototypees como la "receta" para crear un objeto. En cierto modo, la receta por Birdahora incluye todos los "ingredientes" clave de Animal.

**let duck = new Bird("Donald");**

**duck.eat();**

duckhereda todas las Animalpropiedades de , incluido el eatmétodo.

**function Animal() { }**

**Animal.prototype = {**

**constructor: Animal,**

**eat: function() {**

**console.log("nom nom nom");**

**}**

**};**

**function Dog() { }**

**// Only change code below this line**

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

**let beagle = new Dog();**

**Restablecer una propiedad de constructor heredada**

Cuando un objeto hereda su propiedad prototypede otro objeto, también hereda la propiedad constructora del supertipo.

Aquí hay un ejemplo:

**function Bird() { }**

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

**let duck = new Bird();**

**duck.constructor**

Pero ducky todas las instancias de Birddeben mostrar que fueron construidas por Birdy no Animal. Para hacerlo, puede establecer manualmente la propiedad del constructor Birddel Birdobjeto:

**Bird.prototype.constructor = Bird;**

**duck.constructor**

**function Animal() { }**

**function Bird() { }**

**function Dog() { }**

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

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

**// Only change code below this line**

**Bird.prototype.constructor = Bird;**

**Dog.prototype.constructor = Dog;**

**let duck = new Bird();**

**let beagle = new Dog();**

**Agregar métodos después de la herencia**

Una función constructora que hereda su prototypeobjeto de una función constructora de supertipo aún puede tener sus propios métodos además de los métodos heredados.

Por ejemplo, Birdes un constructor que hereda su prototypede Animal:

**function Animal() { }**

**Animal.prototype.eat = function() {**

**console.log("nom nom nom");**

**};**

**function Bird() { }**

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

**Bird.prototype.constructor = Bird;**

Además de lo que se hereda de Animal, desea agregar un comportamiento que sea exclusivo de Birdlos objetos. Aquí, Birdobtendrá una fly()función. Las funciones se agregan de Bird's prototypela misma manera que cualquier función constructora:

**Bird.prototype.fly = function() {**

**console.log("I'm flying!");**

**};**

Ahora las instancias de Birdtendrán ambos eat()y fly()métodos:

**let duck = new Bird();**

**duck.eat();**

**duck.fly();**

duck.eat()mostraría la cadena nom nom nomen la consola y duck.fly()mostraría la cadena I'm flying!.

**function Animal() { }**

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

**function Dog() { }**

**// Only change code below this line**

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

**Dog.prototype.constructor = Dog;**

**Dog.prototype.bark = function() {**

**console.log("Woof!")**

**}**

**// Only change code above this line**

**let beagle = new Dog();**

**Anular métodos heredados**

En lecciones anteriores, aprendiste que un objeto puede heredar su comportamiento (métodos) de otro objeto haciendo referencia a su prototypeobjeto:

**ChildObject.prototype = Object.create(ParentObject.prototype);**

Luego, ChildObjectrecibió sus propios métodos encadenándolos en su prototype:

**ChildObject.prototype.methodName = function() {...};**

Es posible anular un método heredado. Se hace de la misma manera: agregando un método para ChildObject.prototypeusar el mismo nombre de método que el que se va a anular. Aquí hay un ejemplo de Birdcómo anular el eat()método heredado de Animal:

**function Animal() { }**

**Animal.prototype.eat = function() {**

**return "nom nom nom";**

**};**

**function Bird() { }**

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

**Bird.prototype.eat = function() {**

**return "peck peck peck";**

**};**

Si tiene una instancia let duck = new Bird();y llama duck.eat(), así es como JavaScript busca el método en la prototypecadena de duck:

duck=> ¿Se eat()define aquí? No.

Bird=> ¿Se eat()define aquí? => Sí. Ejecútalo y deja de buscar.

Animal=> eat()también está definido, pero JavaScript dejó de buscar antes de alcanzar este nivel.

Objeto => JavaScript dejó de buscar antes de alcanzar este nivel.

**function Bird() { }**

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

**function Penguin() { }**

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

**Penguin.prototype.constructor = Penguin;**

**// Only change code below this line**

**Penguin.prototype.fly = function() {**

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

**};**

**// Only change code above this line**

**let penguin = new Penguin();**

**console.log(penguin.fly());**

**Use un Mixin para agregar un comportamiento común entre**

**objetos no relacionados**

Como has visto, el comportamiento se comparte a través de la herencia. Sin embargo, hay casos en los que la herencia no es la mejor solución. La herencia no funciona bien para objetos no relacionados como Birdy Airplane. Ambos pueden volar, pero a Birdno es un tipo de Airplaney viceversa.

Para objetos no relacionados, es mejor usar mixins . Un mixin permite que otros objetos usen una colección de funciones.

**let flyMixin = function(obj) {**

**obj.fly = function() {**

**console.log("Flying, wooosh!");**

**}**

**};**

El flyMixintoma cualquier objeto y le da el flymétodo.

**let bird = {**

**name: "Donald",**

**numLegs: 2**

**};**

**let plane = {**

**model: "777",**

**numPassengers: 524**

**};**

**flyMixin(bird);**

**flyMixin(plane);**

Aquí birdy planese pasan a flyMixin, que luego asigna la flyfunción a cada objeto. Ahora birdy planeambos pueden volar:

**bird.fly();**

**plane.fly();**

La consola mostraría la cadena Flying, wooosh!dos veces, una para cada .fly()llamada.

Observe cómo el mixin permite que el mismo flymétodo sea reutilizado por objetos no relacionados birdy plane.

**let bird = {**

**name: "Donald",**

**numLegs: 2**

**};**

**let boat = {**

**name: "Warrior",**

**type: "race-boat"**

**};**

**// Only change code below this line**

**let glideMixin = function(obj) {**

**obj.glide = function() {**

**conaole.log("gliding!")**

**}**

**};**

**glideMixin(bird);**

**glideMixin(boat);**

**Use el cierre para proteger las propiedades dentro de un objeto para que no se modifiquen externamente**

En el desafío anterior, birdtenía una propiedad pública name. Se considera público porque se puede acceder a él y modificarlo fuera de la birddefinición de .

**bird.name = "Duffy";**

Por lo tanto, cualquier parte de su código puede cambiar fácilmente el nombre de birda cualquier valor. Piense en cosas como contraseñas y cuentas bancarias que se pueden modificar fácilmente en cualquier parte de su base de código. Eso podría causar muchos problemas.

La forma más sencilla de hacer que esta propiedad pública sea privada es creando una variable dentro de la función constructora. Esto cambia el alcance de esa variable para que esté dentro de la función constructora en lugar de estar disponible globalmente. De esta manera, solo se puede acceder a la variable y cambiarla mediante métodos también dentro de la función constructora.

**function Bird() {**

**let hatchedEgg = 10;**

**this.getHatchedEggCount = function() {**

**return hatchedEgg;**

**};**

**}**

**let ducky = new Bird();**

**ducky.getHatchedEggCount();**

Aquí getHatchedEggCounthay un método privilegiado, porque tiene acceso a la variable privada hatchedEgg. Esto es posible porque hatchedEggse declara en el mismo contexto que getHatchedEggCount. En JavaScript, una función siempre tiene acceso al contexto en el que fue creada. Esto se closurellama

**function Bird() {**

**let weight = 15;**

**this.getWeight = function() {**

**return weight;**

**};**

**}**

**Comprender la expresión de función inmediatamente invocada (IIFE)**

Un patrón común en JavaScript es ejecutar una función tan pronto como se declara:

**(function () {**

**console.log("Chirp, chirp!");**

**})();**

Esta es una expresión de función anónima que se ejecuta de inmediato y se genera Chirp, chirp!de inmediato.

Tenga en cuenta que la función no tiene nombre y no se almacena en una variable. Los dos paréntesis () al final de la expresión de la función hacen que se ejecute o invoque inmediatamente. Este patrón se conoce como una expresión de función invocada inmediatamente o IIFE .

**(function () {**

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

**})();**

**Use un IIFE para crear un módulo**

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

**function glideMixin(obj) {**

**obj.glide = function() {**

**console.log("Gliding on the water");**

**};**

**}**

**function flyMixin(obj) {**

**obj.fly = function() {**

**console.log("Flying, wooosh!");**

**};**

**}**

Podemos agrupar estos mixins en un módulo de la siguiente manera:

**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!");**

**};**

**}**

**}**

**})();**

Tenga en cuenta que tiene una expresión de función invocada inmediatamente (IIFE) que devuelve un objeto motionModule. Este objeto devuelto contiene todos los comportamientos de mezcla como propiedades del objeto. La ventaja del patrón de módulo es que todos los comportamientos de movimiento se pueden empaquetar en un solo objeto que luego pueden usar otras partes de su código. Aquí hay un ejemplo usándolo:

**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");**

**};**

**}**

**};**

**})();**