OBJETOS

Se puede crear un objeto iniciandolo directamente:

let objetito = {

nombre: “Objetoo”,

color: “Es dificil de explicar”

}

Puedo agregarle otra propiedad:

objetito.tercerAtributo = 3;

Y acceder al valor de una propiedad:

objetito.tercerAtributo; Devolvera 3

FUNCIONES CONSTRUCTORES

Seria el molde para crear objetos nuevos que compartan caracteristicas predefinidas. Es un modelo para nuevos objetos.

function Modelo1(parametro1, parametrox) {

this.name = parametro1;

this.tiempo = parametrox;

}

Puede tener parametros o no. El utilizar parametros para asignar valores a los atributos hace que el constructor sea mas versatil y util en muchos casos.

INSTANCIAR UN CONSTRUCTOR

Esto sucede cuando se crea una variable con new:

let resultante = new Modelo1(“velocidad”, “minutos”);

Creara un objeto nuevo que vendra con los atributos name y tiempo con los valores asignados de “velocidad” y “minutos” respectivamente.

CORROBORAR INSTANCIAS

Si creo un objeto con new, ese objeto es una instancia de su constructor. Puedo corroborar si un objeto pertenece a una creacion hecha por determinado constructor con:

resultante instanceof Modelo1;

Esto devolvera true.

PROPIEDADES DIRECTAS

Siguiendo con el constructor Modelo1, sus propiedades directas serian name y tiempo, esto porque se definen directamente con la instancia del objeto.

PROPIEDADES PROTOTYPE

Imaginando que todos los objetos creados a partir de Modelo1 vayan a tener el mismo valor en una tercera propiedad direccion, si creara cientos de objetos a partir de Modelo1, habrian cientos de propiedades duplicadas.

Para evitar esto, se le agregaria:

Modelo1.prototype.direccion = “adelante”;

Ahora cada vez que se instancie un objeto con Modelo1, en time tendra el valor de 2.

El prototype, es una parte del constructor de Modelo1.

Tengo entonces dos tipos de propiedades:

Propiedades directas: se definen directamente en la propia instancia del objeto.

Propiedades prototype: que se definen en el prototype.

FUNCION PARA AGREGAR EN ARREGLOS SEPARADOS

let ownProps = [];

let prototypeProps = [];

for (let property in beagle) {

  if (beagle.hasOwnProperty(property)) {

    ownProps.push(property);

  } else {

    prototypeProps.push(property);

  }

}

PROPIEDAD CONSTRUCTOR

Por cada instancia con Modelo1, existe una propiedad especial llamada constructor, esta hace referencia a la funcion constructor que creo la instancia (la function Modelo1(parametro1, parametrox);)

Es util para averiguar que tipo de objeto es un determinado objeto:

resultado.construcor === Modelo1; devolveria true

NOTA: Si la propiedad constructor es sobreescrita (que se puede), este metodo no funcionaria. Por lo que es mejor utilizar instanceof.

AGREGAR VARIAS FUNCIONES AL PROTOTYPE

Lo que aparentemente seria una de las formas, es a traves de establecerle directamente un prototype a un objeto que ya contenga esas propiedades:

Modelo1.prototype = {

variables: 5,

aumentarVelocidad: function() {

this.velocidad++;

}

disminuirVelocidad: function() {

this.velocidad--;

}

}

ESTABLECER LA PROPIEDAD CONSTRUCTOR AL CAMBIAR EL PROTOTYPE

El efecto secundario de ajustar manualmente el prototipo, es que elimina la propiedad constructor.

resultado.constructor === Modelo1; retorna false

resultado instanceof Modelo1; retorna true

Para solucionarlo, cada vez que un prototipo se establece de forma manual a un nuevo objeto, se define la propiedad constructor:

Modelo1.prototype = {

constructor: Modelo1,

variables: 5,

aumentarVelocidad: function() { ... } }

LUGAR DE DONDE PROVENIENE EL PROTOTYPE

El prototype de un objeto, viene del constructor que lo creo, este lo HEREDA.

function Bird(name) {

this.name = name;

}

let duck = new Bird(“Donald”);

duck hereda su prototype de la funcion constructor Bird. Lo puedo corroborar con isPrototypeOf:

Bird.prototype.isPrototypeOf(duck);

1) Todos los objetos en JavaScript (salvo algunas excepciones) tienen un prototype.

2) El prototype de un objeto, ES en si mismo un objeto.

3) Y como este prototype es un objeto, puede tener su propio prototype

=> En el caso de Bird, el prototype de Bird.prototype es Object.prototype:

Object.prototype.isPrototypeOf(Bird.prototype);

Devuelve true.

1) El metodo hasOwnProperty(), es definido en el Object.prototype por ejemplo.

2) Ese metodo se le puede acceder con Bird.prototype.

3) Al que se le puede acceder con duck.

Bird es el supertype de duck, mientras que duck el subtype.

Object es un supertype de todos los objetos en JavaScript, por lo que cualquier objeto puede utilizar hasOwnProperty.

HERENCIAS

*Recuerda el DRY (Don’t Repeat Yourself)*

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);

}

};

describe se repite, por lo que podria crear un supertype (padre) para ambos:

function Animal() { };

Animal.prototype = {

constructor: Animal,

describe: function() {

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

}

};

Y ahora mis dos objetos iniciales quedarian como:

Bird.prototype = {

constructor: Bird

};

Dog.prototype = {

constructor: Dog

};

Ahora, para reutilizar el metodo creado en Animal dentro de Bird y Dog utilizo la HERENCIA

Se debe utilizar esta sintaxis para que herede el prototype de su padre:

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

NOTA: Object.create(obj) crea un nuevo objeto y establece obj como el prototype del nuevo objeto.

NOTAII: Al establecer el prototype de animal como el prototype de Animal, se le da a la instancia animal la misma receta que a cualquier otra instancia de Animal.

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

NOTA III: Ahora la receta Bird contiene todos los ingredientes clave de Animal.

let duck = new Bird("Donald");

duck.eat();

NOTA IV: Ahora duck hereda todas las propiedades de Animal, incluyendo el metodo eat.

Cuando un objeto hereda el prototype de otro objeto, tambien hereda la propiedad del constructor del supertipo.

En el caso de Bird, las instancias deberian mostrar que fueron construidas por Bird y no Animal. La solucion:

Bird.prototype.constructor = Bird;

duck.constructor

Si quisiera agregar un metodo exclusivo para objetos creados con Bird:

Bird.prototype.fly = function() {

document.write(“I’m flying!”);

}

Ahora por cada instancia de Bird tendra el metodo eat() y el metodo fly()

SOBREESCRIBIR UN METODO HEREDADO

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

En el caso de Bird, habia heredado un metodo de Animal que era eat().

Para modificarlo:

Bird.prototype.eat = function() {

return "peck peck peck";

};

*COLORARIO:*

*Si tienes una instancia de let duck = new Bird(); y llamas a duck.eat(), así es como JavaScript busca el método en la cadena prototype de duck:*

1. *duck => ¿Está eat() definido aquí? No.*
2. *Bird => ¿Está eat() definido aquí? => Sí. Ejecútala y detén la búsqueda.*
3. *Animal => eat() también está definido, pero JavaScript dejó de buscar antes de llegar a este nivel.*
4. *Object => JavaScript dejó de buscar antes de llegar a este nivel.*

MIXIN

Objetos no relacionados podrian compartir metodos, pero estos al no proceder de iguales constructores, no deberian instanciarse bajo un mismo constructor. Por ejemplo un pajaro y un avion.

let flyMixin = function(obj) {

obj.fly = function() {

console.log(“Flying, woooosh!”);

}

}

Al realizar una instancia de un constructor Plan:

let plane = new Plane();

flyMixin(plane); le agrega el metodo fly a plane.

PROPIEDADES PUBLICAS

duck tenia una propiedad publica, name.

duck.name = “otro nombre”;

Esta propiedad se la puede cambiar muy facilmente. Si fuera una contrasenia eso seria un problema.

PROPIEDADES PRIVADAS

Para modificar una propiedad de publica a privada se crea una variable dentro de la funcion constructora. De esta forma la variable solo puede ser accesible y cambiable por metodos que esten tambien dentro de la funcion constructora.

Las variables privadas no se declaran con this, se declaran con let, a la cual luego se le accede con un metodo en el mismo constructor.

function Bird() {

let hatchedEgg = 10;

this.getHatchedEggCount = function () {

return hatchedEgg;

};

}

AGRUPAR NUEVOS METODOS EN UNA IIFE

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!");

};

}

}

})();