**OBJETOS EN JAVASCRIPT**

Javascript es un lenguaje orientado a objetos con las funcionalidades de un lenguaje funciona asíncrono. Como todo lenguaje orientado a objetos, su funcionamiento estará basado en su totalidad por objetos.

Un objeto es una colección de propiedades. Cada propiedad es una asociación entre un nombre y un valor. Ejemplo: **let objeto = { nombre: valor }** .

Está encerrado entre corchetes { } y hacemos la asociación del nombre con el valor con : { nombre : valor }.

Dentro de un objeto podemos asignar todo tipo de variables y datos, como funciones. Las **funciones que se asignan dentro de un objeto** tienen el nombre de **métodos**.

Ejemplo de un método:

**let objeto = {**

**metodo: function (){ return “metodo” }**

**}**

Este método luego lo podemos ejecutar utilizando el objeto que contiene ese método. Lo llamamos tal que así: **objeto.metodo()**

Si queremos ir añadiendo distintas propiedades solo tenemos que ir separándolas con comas, ¡pero ojo!, la última propiedad no puede contener ni una coma o punto y coma, o generará un error.

Ejemplo de objeto con varias propiedades y métodos:

**let objeto = {**

**numero: 4,**

**string: “esto es un string”,**

**boleano: true,**

**metodo: () => console.log(“hello”),**

**otroObjeto: { num: 5 }**

**}**

**Prototipos en Javascript**

Los objetos se vinculan con otros objetos a través de prototype. Cuando tenemos varios objetos que compartan la misma estructura, podemos crear un prototipo para crear más objetos con la misma estructura (instancia).

El prototipo de una instancia se puede visualizar y modificar con **instancia.\_\_proto\_\_** .

La diferencia entre prototipos y clases, es que si modificamos el prototipo desde cualquiera de los objetos que compartan el mismo prototipo, no solo se modificará ese mismo objeto sino todos los que compartan el prototipo. Puede resultar confuso pero es una de las características más potentes de Javascript.

function Punto(x, y) {

this.x = x,

this.y = y,

this.metodo = (string) => console.log(string);

}

Con el “**this**”, estamos apuntando al objeto instanciado en concreto. Por lo tanto cuando creemos una nueva instancia de este objeto, nos estaremos refiriendo a ese nuevo objeto.

Ahora, si queremos crear una **instancia de ese prototipo para crear un objeto** del tipo del objeto padre, solo utilizamos la siguiente nomenclatura: **let instancia = new Objeto();**

Con esto alojamos el nuevo objeto con la palabra clave new, en una variable para poder utilizarla cuando y como queramos.

**let p1 = new Punto(0, 4);**

**let p2 = new Punto(2, 3);**

Ya tenemos nuestra instancia del objeto pero queremos añadir una nueva funcionalidad o una nueva propiedad. Aquí es donde entra el potencial de prototype, para modificar el objeto padre (prototipo) y sus instancias. Tenemos dos formas de utilizar prototype: Podemos modificar el propio prototipo (objeto padre) con el método **prototype**, o bien podemos modificar el prototipo desde cualquier instancia de ese objeto con el método **\_\_proto\_\_**.

Ejemplos:

* Desde el objeto padre (prototipo):

**Punto.prototype.imprimir = function() {**

**console.log (`X del punto: ${this.x}. \nY del punto: ${this.y}.`);**

**}**

* Desde cualquier instancia del objeto:

**p1.\_\_proto\_\_.imprimir = function(paraImprimir){**

**console.log (`X del punto: ${this.x}. \nY del punto: ${this.y}.`);**

**}**

Ahora ya podemos acceder al nuevo método del objeto en cualquier instancia hecha o por hacer: **p2.imprimir();**

Devuelve en consola -> X del punto: 2. Y del punto: 3.

Así pues, para dejar claro que nos permiten hacer los prototipos: Nos permiten crear objetos que heredan las mismas propiedades y los mismos métodos y, luego modificar el objeto prototipo para que tanto el prototipo como sus instancias se modifiquen.

La diferencia clara entre las clases, es que si en una clase creas una instancia y luego modificas esa misma clase, las instancias ya creadas no serán modificadas ni actualizadas.

En Javascript existen las clases desde ECMASCript 2015, utilizando la palabra clave class, pero realmente no son clases sino que son una forma simple y clara de crear prototipos y objetos. Por lo tanto, **el funcionamiento de los prototipos y de las clases son exactamente el mismo, sencillamente cambia como los declaramos**.

Entender cómo funciona la herencia y el prototype en Javascript, es lo más fundamental para entender el propio lenguaje. En cuanto herencia se refiere, Javascript sólo tiene una estructura: objetos.

Cada objeto tiene una propiedad privada (Prototype) que mantiene un enlace a otro objeto del que hereda (su prototipo). Esto es lo más importante e imprescindible de entender, en Javascript todos los datos, objetos y funciones heredan de otro objeto, y ese objeto hereda de otro objeto, así de forma contínua hasta llegar al objeto null que no tiene prototype y que podríamos definir como objeto inicial y el final de la cadena de prototipos.

**let Objeto1 = {a: 1};**

// Cadena de prototipado: Objeto1 -> Object.prototype -> null

**let array = [“algo”, “string”, 3];**

// Cadena de prototipado: array -> Array.prototype -> Object.prototype -> null

**function funcion(){return true}**

// Cadena de prototipado: funcion -> Function.prototype -> Object.prototype -> null

**Maneras diferentes de crear objetos**

En Javascript tenemos múltiples formas de crear objetos.

* **Mediante la notación literal (Inicializador de objeto):**

**let objeto = {**

**propiedad\_1 : valor\_1,**

**propiedad\_2: valor\_2,**

**“propiedad n”: valor\_n**

**}**

// Cadena de prototipado: objeto -> Object.prototype -> null

* **Mediante una función constructora (función prototipo):**

**function Coche(marca, modelo, anno) {**

**this.marca = marca,**

**this.modelo = modelo,**

**this.anno = anno;**

**}**

// Cadena de prototipado: Coche -> Object.prototype -> null

**let audiR8 = new Coche(“Audi”, “R8”, “2010”);**

// Cadena de prototipado: audiR8 -> Coche.prototype -> Object.prototype -> null

Esta es la forma que hemos visto antes de construir un objeto padre prototipo y por consiguiente podemos instanciar ese objeto con el operador/palabra clave **new**.

Desde ECMAScript 2015 **podemos utilizar como función constructora las clases**, que como ya hemos dicho antes es exactamente lo mismo que una función constructora prototipo solo que con una sintaxis más clara y sencilla.

* **Mediante el método Object.create()**

Este método es bastante útil si queremos elegir un objeto sin función constructora como objeto prototipo de nuestra nueva función. Por lo tanto podremos crear una instancia de un objeto creado con notación literal y que herede de ese objeto sin función constructora.

**let objeto = {a: 1}**

// Cadena de prototipado: objeto -> Object.prototype -> null

**let nuevoObjeto = Object.create(objeto);**

// Cadena de prototipado: nuevoObjeto -> objeto.prototype -> Object.prototype -> null

También podemos crear un objeto con notación literal con un método constructor y crear la instancia con Object.create(). Así podremos inicializar un objeto heredado con notación literal e inicializar propiedades únicas para ese objeto hijo (instancia).

**let Objeto = {**

**init: function init(x, y) {**

**this.x = x,**

**this.y = y;**

**},**

**moverX: function(x){**

**this.x += x;**

**}**

**}**

**const punto1 = Object.create(Objeto);**

**punto1.init(3, 2);**

Así hemos heredado punto1 del objeto Objeto, y además hemos inicializado punto1 para asignar las propiedades de punto1. Si no añadimos un método constructor, no podremos inicializar la instancia con parámetros personalizados, tan solo heredaremos los datos que tengamos en el otro objeto.

Un atajo para evitar tener que escribir el Object.create() manualmente, es añadirlo dentro del mismo método constructor. Tal que así:

**let Objeto = {**

**init: function init(x, y) {**

**let obj = Object.create(this)**

**obj.x = x,**

**obj.y = y;**

**return obj;**

**},**

**moverX: function(x){**

**this.x += x;**

**}**

**}**

**let punto1 = Punto.init(3, 2);**

Construir objetos con Object.create() puede ser bastante complejo y su función va más allá de poder instanciar objetos.

* **Mediante el uso de clases**

Como hemos mencionado antes, las clases no existen en Javascript. Las clases de Javascript son funciones prototipos maquillados para que la sintaxis sea más sencilla y legible, por lo tanto se comportan exactamente igual.

La diferencia de sintaxis es radicalmente diferente y realmente es mucho más legible que una función prototipo o un objeto con notación literal y función constructora.

Para crear una “clase” escribiremos la palabra clave **class** seguido del nombre de la clase. Tal y como hacemos con los objetos prototipos, es una buena práctica escribir la primera letra de la “clase” en mayúsculas. Luego abrimos con corchetes y allí podemos incluir tanto la función constructora, como el resto de métodos de nuestro objeto.

**class Punto {**

**constructor(x, y) {**

**this.x = x,**

**this.y = y;**

**}**

**moverEnX(x) {**

**this.x += x;**

**}**

**}**

**const p1 = new Punto(3, 2);**

Así por lo tanto podemos añadir todas las propiedades y métodos que queramos de una forma mucho más limpia y legible. Ahora bien, recuerda siempre que son funciones prototipo solo que con una sintaxis más bonita. Para entender como funcionan las clases, es imprescindible entender las funciones prototipo y como se modifican esos prototipos.