CONTENIDO:

Programación orientada a objetos en JavaScript.

1. INTRODUCCIÓN:

JavaScript es un lenguaje ***multiparadigma***, es decir JavaScript puede usar varias metodologías de programación como la programación funcional o programación orientada a objetos.

La programación orientada a objetos (POO) es un modelo o filosofía de lenguaje basado en el concepto de objeto, el cual puede contener información (propiedades o atributos) y un comportamiento (métodos o funciones), de modo muy similar a los objetos del mundo verdadero

Por tanto, la POO es un paradigma de programación muy utilizado en muchos lenguajes de programación pero que en JavaScript toma importancia a partir de la ECMAScript 6.

En la POO todo sería un objeto como hemos dicho. Por tanto, lo primero en preguntarnos sería: ¿qué sería un objeto?: Pues un objeto es una abstracción del mundo real, es decir, sería una representación del estado y las acciones que puede realizar algo/alguien.

En la POO en General tenemos cuatro conceptos muy importantes:

a) ***Clases***. Modelo a seguir. Sería algo sobre lo que basarnos. Un molde o patrón que define qué propiedades y métodos va a tener un objeto que sea una instancia de esta clase.
b) ***Objetos***. Instancia de una clase.
    a. Atributos. Característica o propiedad que posea el objeto. En los lenguajes de programación serían las variables.
    b. Métodos. Son las acciones que un objeto puede realizar. (Todos los métodos serían verbos ej: getElementById). Los métodos serían funciones dentro de un objeto.

Un lenguaje de POO puede basarse en clases o en prototipos. En el primer caso tenemos ejemplos como Java, C ++ o Php, mientras que en el segundo tenemos otros como Javascript, que es el que nos interesa.

Por tanto, JavaScript es un lenguaje Orientado a Objetos, pero **basado en Prototipos** no en clases. Cuando creamos una clase en JavaScript, el motor de JavaScript (navegador, servidor, etc) lo que realmente está haciendo es convertir o transformar esa clase en una función prototipal. Así, el concepto de prototipo en JavaScript adquiere mucha importancia.

En Javascript no tenemos clases (aunque en ES6 ya podemos usar la sintaxis de class y que veremos posteriormente), únicamente existen los objetos; ***"todo es un objeto"*** . En su lugar, un objeto logra la herencia por medio del atributo prototype, una referencia a otro objeto, su prototipo.

Por ese motivo, voy a explicar con detalle el prototipado de JavaScript:

Se puede consultar el enlace developer.mozilla.org para más detalle.

En forma esquematizada sería:

Programación orientada a Objetos

- Es un PARADIGMA de programación
- Es el más usado hoy en día
- Aplicable a muchísimos lenguajes

Paradigma

- Una forma de pensar o actuar, una filosofía
- Una serie de patrones o modelos a seguir
- Un planteamiento común

¿Qué es un objeto?

- Es una abstracción del mundo real
- Es una representación del estado y las acciones que puede realizar algo/alguien

Atributos

- Características del objeto
- Se refieren al estado actual
- Son modificables y accesibles

Métodos

- Acciones del objeto
- Se refieren al estado actual
- Son modificables y accesibles

Prototipos

- Es la base de toda la programación en JS.
- Ya que todo es un objeto, a la vez todo debe partir de un prototipo.

this (Scope)

- A partir de ES6 el alcance del this se limita al objeto declarado.
- De una forma global, el this sería igual a window

2. PROTOTIPOS EN JAVASCRIPT

Un prototipo en JavaScript es un mecanismo por el cual un objeto puede heredar de un objeto padre atributos y métodos.

```
// función constructora.
function  Animal(nombre, genero) {
    // this dentro de una función constructora hace referencia a la propia función.
    // los atributos y métodos deben de colgar de this.
        // Atributos:
    this.nombre = nombre;
    this.genero = genero;
        // Métodos:
    this.saludar = function () {
        console.log(`Hola ${this.nombre} `);
    }
}
```

```
// Para crear instancias:
const animal1 = new Animal("perro", "macho"),
animal2 = new Animal("gato", "hembra");

console.log(animal1);
animal1.saludar();
console.log(animal2);
// Recuerdo que en una Arroy Function, this hace referencia a Windows. Mucho cuidado!!
```

Este código repetiría la función saludo tantas veces como objetos o prototipos me cree. (no muy eficiente). Si nos fijamos vemos que por cada objeto Animal, además de los elementos de construcción (nombre,genero) también replicamos el método saludar.

Animal {nombre: 'perro', genero: 'macho', saludar: ƒ}

1. **genero**: "macho"
2. **nombre**: "perro"
3. **saludar**: *ƒ ()*
4. [[Prototype]]: Object
    1. **constructor**: *ƒ Animal(nombre, genero)*
    2. [[Prototype]]: Object

Por tanto, una mejora sería asignar los métodos al prototipo y sacarlos de la función constructora. Usaremos ***NombreFuncionPrototipo.prototype.metodo:***

```
// función constructora.
function  Animal(nombre, genero) {
    // this dentro de una función constructora hace referencia a la propia función.
    // los atributos y métodos deben de colgar de this.
        // Atributos:
    this.nombre = nombre;
    this.genero = genero;
        // Métodos:
    }
// Los métodos los convertimos en métodos prototípicos que no se duplicarán
// cuando creemos más objetos.
Animal.prototype.saludar = function () {
    console.log(`Hola ${this.nombre} `);
}
// Para crear instancias :
const animal1 = new Animal("perro", "macho"),
animal2 = new Animal("gato", "hembra");

console.log(animal1);
animal1.saludar();
```

```
console.log(animal2);
```

Se puede ver que ahora el método saludar pertenece al prototype y por tanto no se duplica por cada objeto Animal que creemos.

Animal {nombre: 'perro', genero: 'macho'}

1. **genero**: "macho"
2. **nombre**: "perro"
3. [[Prototype]]: Object
    1. **saludar**: *ƒ ()*
    2. **constructor**: *ƒ Animal(nombre, genero)*
    3. [[Prototype]]: Object

3. HERENCIA DE PROTOTIPOS O HERENCIA PROTOTÍPICA EN JAVASCRIPT

Sería la capacidad de heredar propiedades de un padre a un hijo a través de cadena de herencia prototípica.

```
function Gato(nombre, genero, raza) {
    this.super = Animal;
    this.super(nombre, genero);
    this.raza = raza;
}
// la herencia la realizaría a través de las siguientes 2 líneas:
Gato.prototype = new Animal();
Gato.prototype.constructor = Gato;
// La sobreescritura del método ha de hacerse posteriormente a la creación.
Gato.prototype.saludar = function (mensaje) {
    console.log(`${mensaje} Sobreescribiendo el método saludar del
padre:  ${this.nombre} `);
}
// ahora me creo un objeto de tipo animal.
const gato1 = new Gato("Gato1", "Hembra", "Angora");
console.log(gato1);
// llamo a un método de su padre (Animal)
gato1.saludar("Bienvenido, ");
```

Si nos fijamos en la información de la consola se puede ver que Gato hereda prototípicamente de Animal que a su vez hereda del objeto primitivo Object.

Gato {nombre: 'Gato1', genero: 'Hembra', raza: 'Angora', super: ƒ}

1. **genero**: "Hembra"
2. **nombre**: "Gato1"
3. **raza**: "Angora"
4. **super**: *ƒ Animal(nombre, genero)*
5. [[Prototype]]: Animal
    1. **constructor**: *ƒ Gato(nombre, genero, raza)*
    2. **genero**: undefined
    3. **nombre**: undefined
    4. **saludar**: *ƒ (mensaje)*
    5. [[Prototype]]: Object

4. CLASS A PARTIR DE ECMAScript 6:

JavaScript introdujo la palabra clave **class** en ECMAScript 2015 también llamado ECMAScript 6. Hace que JavaScript ***parezca*** un lenguaje POO. Pero solo es ***azúcar sintáctico***[1] o un edulcorante sobre la técnica de creación de prototipos existente. Internamente el navegador o el servidor siguen trabajando con la creación de prototipos en segundo plano, pero hace que el cuerpo exterior parezca POO.

Hasta este punto podemos notar lo siguiente:

- Se introduce la primera palabra reservada de ES6 class para hacer la declaración de la clase.
- Por convención, el nombre de las clases se escribe en PascalCase o también llamada UpperCamelCase.
- Si la clase es creada con class, esta no puede ser llamada como una función: User().
- Automáticamente, el código que está dentro de una clase, es ejecutada en strict mode[2].
- Las clases no obedecen al hoisting, por lo tanto, no puede ser usada antes de su declaración.
- De manera implícita las clases se comportan como constantes, es decir, no puede redeclararse bajo un mismo ámbito.
- La forma de instanciar una clase es a través de new.

Más información en w3schools.com .

```
class Animal {
    //el constructor es un método especial que se ejecuta en el momento de instanciar
la clase
    constructor(nombre, edad, genero) {
      this.nombre = nombre;
      this.genero = genero;
    }
    // Métodos públicos de la clase o también llamados métodos de clase se crean de
la misma forma que los métodos de objeto. Recuerda que JS no tiene métodos privados,
pero sí estáticos que veremos más adelante.
    saludar() {
        console.log(`Hola ${this.nombre} `);
    }
}
const loroLucy = new Animal("Lucy","hembra");
loroLucy.saludar();
```

1 El término fue acuñado en 1964 por Albert J. Landin y se refiere a todas esas construcciones que no aportan nueva funcionalidad a un lenguaje, pero que permiten que sea más fácilmente utilizable por seres humanos.

[2] El propósito de "use strict" es indicar que el código ha de ejecutarse en modo estricto, es decir, no se pueden utilizar variables que no hayan sido declaradas con anterioridad.

Para aplicar herencia usaremos la palabra extends :

```
// con la palabra extends Perro hereda de Animal.
class Perro extends MiAnimal {
    constructor (nombre, tipo, genero, raza) {
        super(nombre,tipo,genero);
        this.raza = raza;
    }
    // un método estático es aquél que se puede ejecutar sin necesidad de instanciar
la clase.
    static comunicar() {
        console.log("Hola soy un perro...");
    }
}
const perro1 = new Perro("Tino","perro","Macho","Pastor Alemán");
Perro.comunicar();
perro1.saludar();
```

Otros ejemplos:

```
class Usuario {
    constructor(nombre, apellido, correo, edad){
        this.nombre = nombre
        this.apellido = apellido
        this.correo = correo
        this.edad = edad
    }
}

let beto = new Usuario("Beto", "Quiroga", "beto@ed.team", 28)
```

```
class Usuario {
    constructor(nombre, apellido, correo, edad){
        this.nombre = nombre
        this.apellido = apellido
        this.correo = correo
        this.edad = edad
    }

    saludar() {
        return document.write(`
            <div>
                <p>
                Hola, mi nombre es ${this.nombre}
                </p>
            </div>
        `)
    }
}

let beto = new Usuario("Beto", "Quiroga", "beto@ed.team", 28)
```

```
class Profesor extends Usuario{
    constructor(nombre, apellido, correo, edad, experiencia, lenguaje){
        super(nombre, apellido, correo, edad)
        this.experiencia = experiencia
        this.lenguaje = lenguaje
    }
}

class Estudiante extends Usuario{
    constructor(nombre, apellido, correo, edad, activado){
        super(nombre, apellido, correo, edad)
        this.activado = activado
    }
}
```

```
class Forma {
    constructor(alto, ancho, color){
        this.alto = alto
        this.ancho = ancho
        this.color = color
    }

    dibujar(){
        return document.body.innerHTML = `
        <div
            style="
            width: ${this.ancho}px;
            height: ${this.alto}px;
            background: ${this.color}"
        >
        </div>
        `
    }
}

let forma1 = new Forma(220, 400, "red")
```

5. SETTERS Y GETTERS:

Una función que obtiene un valor de una propiedad se llama ***getter*** y una que establece el valor de una propiedad se llama ***setter***.

Son, en esencia, funciones que se ejecutan para obtener (“get”) y asignar (“set”) un valor, pero que para un código externo parecen propiedades normales.

Esta característica ha sido implementada en ES2015, pudiendo modificar el funcionamiento normal de establecer u obtener el valor de una propiedad, a estas se les conoce como accessor properties.

```
    get getRaza(){
        return this.raza;
    }
    set setRaza(nuevaRaza){
        this.raza = nuevaRaza;
    }
```

Es importante indicar que para llamar a los set y get usaremos:

perro1.getRaza // (sin paréntesis) nos mostraría la raza del perro1.

perro1.setRaza = “Labrador” // para cambiar la raza del perro

Más información en es.javascript.info.

Existe una ***refactorización*** de código para poder usar como nombre get el propio nombre del atributo y no entrar en problemas. Si no lo refactorizamos entraríamos en un bucle infinito, generando un error. Sería añadir un guion bajo delante de cada propiedad:

```
class Perro extends MiAnimal {
    constructor (nombre, tipo, genero, raza) {
        super(nombre,tipo,genero);
        this._raza = raza;
    }
    // un método estático es aquél que se puede ejecutar sin necesidad de instanciar
la clase.
    static comunicar() {
        console.log("Hola soy un perro...");
    }
    get raza(){
        return this._raza;
    }
    set raza(nuevaRaza){
        this._raza = nuevaRaza;
    }
}
```