JavaScript

Programación Orientada a Objetos (POO)

CertiDevs

Índice de contenidos

La **programación orientada a objetos** (OOP) es un paradigma de programación que utiliza objetos y sus interacciones para modelar y organizar la lógica de la aplicación. JavaScript es un lenguaje de programación que soporta la OOP a través de prototipos y clases. A continuación, se detallan más aspectos de la OOP en JavaScript.

1. Prototipos

Antes de la introducción de las clases en ECMAScript 2015 (ES6), JavaScript usaba prototipos para lograr la herencia y la OOP. Los objetos en JavaScript tienen una propiedad llamada prototype que es una referencia a otro objeto.

Cuando se busca una propiedad en un objeto, si el objeto no tiene la propiedad, JavaScript buscará en el objeto prototipo y continuará siguiendo la cadena de prototipos hasta que encuentre la propiedad o llegue al final de la cadena.

Para ilustrar cómo funcionan los prototipos, aquí hay un ejemplo de cómo crear una "clase" y una instancia usando la sintaxis de prototipos:

```
function Persona(nombre, edad) {
    this.nombre = nombre;
    this.edad = edad;
}

Persona.prototype.presentarse = function() {
    console.log(`Hola, mi nombre es ${this.nombre} y tengo ${this.edad} años.`);
};
const juan = new Persona('Juan', 28);
juan.presentarse(); // Hola, mi nombre es Juan y tengo 28 años.
```

2. Clases

Las **clases** en JavaScript son una forma de representar objetos y proporcionar una sintaxis más clara y concisa para trabajar con la herencia y la programación orientada a objetos.

Las clases en JavaScript son una capa de azúcar sintáctico sobre la herencia prototípica existente en el lenguaje. Se comportan de forma similar a las clases en otros lenguajes de programación.

2.1. Definición de clases

Para definir una clase en JavaScript, se utiliza la palabra clave `class` seguida del nombre de la clase.

Por convención, los nombres de las clases comienzan con una letra mayúscula.

```
class Persona {
```

```
    // Código de la clase aquí
}
```

2.2. Constructor

El **constructor** es un método especial dentro de una clase que se ejecuta cuando se crea una nueva instancia de la clase. Se define usando la palabra clave `constructor`.

```
class Persona {
    constructor(nombre, edad) {
        this.nombre = nombre;
        this.edad = edad;
    }
}
```

2.3. Atributos públicos

Por defecto, los atributos de una clase en JavaScript son **públicos**, lo que significa que pueden ser accedidos y modificados desde cualquier parte del código.

Los atributos públicos se definen en el constructor de la clase o como campos de clase.

Puedes definir atributos públicos en el constructor de una clase utilizando la palabra clave this y asignándoles un valor.

```
class Persona {
    constructor(nombre, edad) {
        this.nombre = nombre;
        this.edad = edad;
    }
}
```

2.4. Campos de clase

También puedes definir atributos públicos utilizando **campos de clase**. Los campos de clase se declaran dentro del cuerpo de la clase y pueden tener un valor predeterminado.

```
class Persona {
    nombre = 'Anónimo';
    edad = 0;
}
```

2.5. Métodos de instancia

Los métodos de instancia son funciones que pertenecen a las instancias de una clase. Se definen dentro del cuerpo de la clase, **sin la palabra clave function**.

```
class Persona {
    constructor(nombre, edad) {
        this.nombre = nombre;
        this.edad = edad;
    }
    presentarse() {
        console.log(`Hola, mi nombre es ${this.nombre} y tengo ${this.edad} años.`);
    }
}
```

3. Creación de instancias (objetos)

Para crear una nueva instancia de una clase, se utiliza la palabra clave `new` seguida del nombre de la clase y los argumentos del constructor entre paréntesis.

```
const juan = new Persona('Juan', 28);
juan.presentarse(); // Hola, mi nombre es Juan y tengo 28 años.
```

4. Encapsulamiento y propiedades privadas

La **encapsulación** hace referencia a encapsular los atributos de una clase para que no puedan ser modificados directamente desde fuera de la clase y deba usarse un método de la clase. De esta forma el método puede controlar la forma en que se modifican los atributos, añadiendo validaciones, haciendo comprobaciones, etc.

Las **propiedades privadas** son aquellas que solo se pueden acceder y modificar dentro de la clase. En JavaScript, puedes declarar propiedades privadas usando el caracter `#` antes del nombre de la propiedad.

```
class Ejemplo {
    #propiedadPrivada;

    constructor(valor) {
        this.#propiedadPrivada = valor;
    }

    mostrarPropiedadPrivada() {
        console.log(this.#propiedadPrivada);
    }
}
```

```
const instancia = new Ejemplo();
instancia.mostrarPropiedadPrivada();
```

Los **atributos privados** son aquellos que solo se pueden acceder y modificar dentro de la clase en la que se definen. Para declarar un atributo privado en una clase, utiliza el caracter `#` antes del nombre del atributo.

```
class Ejemplo {
    #atributoPrivado = 'valor privado';

    mostrarAtributoPrivado() {
        console.log(this.#atributoPrivado);
    }
}
```

Los atributos privados proporcionan un nivel de encapsulamiento que permite a los desarrolladores controlar el acceso y la modificación de los datos internos de una clase. Esto puede ser útil para mantener la integridad del estado de un objeto y ocultar detalles de implementación.

4.1. Getters y Setters

Los **getters** y **setters** son métodos especiales que permiten obtener y modificar el valor de una propiedad de un objeto de manera controlada. Se utilizan las palabras clave `get` y `set` para definir estos métodos en una clase.

```
class Circulo {
    constructor(radio) {
        this._radio = radio;
    }

    // Getter para calcular el área
    get area() {
        return Math.PI * this._radio ** 2;
    }

    // Getter para calcular el perímetro
    get perimetro() {
        return 2 * Math.PI * this._radio;
    }

    // Setter para modificar el radio
    set radio(nuevoRadio) {
        if (nuevoRadio <= 0) {
            throw new Error('El radio debe ser positivo');
        }
        this._radio = nuevoRadio;
    }
```

```
}

const circulo = new Circulo(5);
console.log(circulo.area); // 78.53981633974483
console.log(circulo.perimetro); // 31.41592653589793
circulo.radio = 10;
console.log(circulo.area); // 314.1592653589793
```

Se utiliza `_` delante del atributo para evitar la recursión infinita, ya que si no se pone `_` entonces al invocar el setter se llama de nuevo al getter.

4.2. Métodos y propiedades privados

Además de las propiedades privadas, también puedes crear **métodos privados** utilizando el caracter # antes del nombre del método. Los métodos privados solo pueden ser invocados desde dentro de la clase.

```
class Ejemplo {
    #metodoPrivado() {
        console.log('Este es un método privado');
    }

    invocarMetodoPrivado() {
        this.#metodoPrivado();
    }
}
const instancia = new Ejemplo();
instancia.invocarMetodoPrivado(); // Este es un método privado
// instancia.#metodoPrivado(); // Error: Método privado no accesible desde fuera de la
clase
```

4.3. Diferencia entre # y _

En JavaScript, el prefijo _ y el prefijo # se utilizan para diferentes propósitos cuando se trata de la encapsulación de atributos.

Prefijo _: El guion bajo _ se utiliza como un convenio para indicar que una propiedad o un método debe ser tratado como "privado" o no accesible directamente desde fuera de la clase. Sin embargo, esta es solo una convención, y en realidad no hace que la propiedad o el método sea privado. Todavía se puede acceder y modificar desde fuera de la clase, pero se considera una mala práctica hacerlo.

```
class Ejemplo {
    constructor(id) {
    this._id = id;
    }
```

```
    get id() {
        return this._id;
    }

    set id(nuevoId) {
        this._id = nuevoId;
    }
}
```

Prefijo #: El símbolo # se utiliza para crear verdaderas propiedades privadas en JavaScript. Las propiedades privadas no pueden ser accedidas ni modificadas desde fuera de la clase, lo que garantiza una encapsulación adecuada.

```
class Ejemplo {
    #id;

    constructor(id) {
        this.#id = id;
    }

    get id() {
        return this.#id;
    }

    set id(nuevoId) {
        this.#id = nuevoId;
    }
}
```

4.4. Campos de clase

Los **campos de clase** son una forma más concisa de declarar propiedades en una clase, sin la necesidad de un constructor explícito.

Estos campos se inicializan automáticamente cuando se crea una nueva instancia de la clase.

```
class Ejemplo {
    propiedad1 = 'valor1';
    propiedad2 = 'valor2';
}
const instancia = new Ejemplo();
console.log(instancia.propiedad1); // valor1
console.log(instancia.propiedad2); // valor2
```

5. Atributos de solo lectura

Para crear un **atributo de solo lectura**, puedes usar un **getter** sin proporcionar un **setter**. Esto significa que el atributo solo se puede acceder, pero no se puede modificar directamente.

```
class Circulo {
    constructor(radio) {
        this._radio = radio;
    }

    // Getter para obtener el radio
    get radio() {
        return this._radio;
    }
// No se proporciona un setter, por lo que el radio es de solo lectura
}
```

6. Atributos estáticos

Los **atributos estáticos** pertenecen a la clase en sí y no a sus instancias.

Para definir un atributo estático en una clase, utiliza la palabra clave `static` antes del nombre del atributo.

```
class Ejemplo {
    static atributoEstatico = 'valor estático';

    mostrarAtributoEstatico() {
        console.log(Ejemplo.atributoEstatico);
    }
}
```

Los atributos estáticos son útiles cuando se necesita compartir información entre todas las instancias de una clase o cuando se quiere almacenar información que no depende del estado de una instancia específica.

7. Métodos estáticos

Los **métodos estáticos** son funciones que pertenecen a la clase en sí y no a sus instancias. Se definen usando la palabra clave static antes del nombre del método.

```
class Matematicas {
    static sumar(a, b) {
        return a + b;
    }
```

```
}

const resultado = Matematicas.sumar(1, 2); // 3
```

8. Herencia

Las clases en JavaScript admiten la **herencia** a través de la palabra clave `extends`. Esto permite que una clase herede propiedades y métodos de otra clase.

```
class Persona {
    constructor(nombre, edad) {
        this.nombre = nombre;
        this.edad = edad;
    }

    presentarse() {
        console.log(`Hola, mi nombre es ${this.nombre} y tengo ${this.edad} años.`);
    }
}

class Empleado extends Persona {
    constructor(nombre, edad, puesto) {
        super(nombre, edad);
        this.puesto = puesto;
    }

    presentarse() {
        super.presentarse();
        console.log(`Trabajo como ${this.puesto}.`);
    }
}
const juan = new Empleado('Juan', 28, 'desarrollador');
juan.presentarse();
// Hola, mi nombre es Juan y tengo 28 años.
// Trabajo como desarrollador.
```

En este ejemplo, la clase Empleado hereda las propiedades y métodos de la clase Persona. La función super se utiliza para llamar al constructor y los métodos de la clase padre.

9. Polimorfismo

El **polimorfismo** es un concepto fundamental en la programación orientada a objetos que permite a los objetos de diferentes clases ser tratados como objetos de una clase común.

Esencialmente, el **polimorfismo** permite que diferentes objetos respondan al mismo método o función de manera diferente, según su tipo o clase. Esto permite diseñar funciones y métodos más genéricos y reutilizables y simplifica la lógica del programa.

El polimorfismo en JavaScript se puede lograr de varias maneras, incluida la herencia, la anulación de métodos y las interfaces.

El polimorfismo simplifica el diseño y la lógica del programa al permitir la reutilización de código y la creación de funciones y métodos más genéricos.

9.1. Polimorfismo a través de la herencia

Una forma común de lograr el **polimorfismo** en JavaScript es a través de la **herencia**, donde una clase hereda propiedades y métodos de otra clase. En este caso, las clases derivadas pueden anular y extender los métodos heredados de la clase base para proporcionar un comportamiento específico de la clase derivada.

Por ejemplo, supongamos que tenemos una clase base Forma y varias clases derivadas como Circulo, Cuadrado y Triangulo, cada una con su propia implementación del método area():

```
class Forma {
    area() {
        throw new Error('El método "area()" debe ser implementado en la clase
derivada.');
    }
}

class Circulo extends Forma {
    constructor(radio) {
        super();
        this.radio = radio;
    }

    area() {
        return Math.PI * this.radio ** 2;
    }
}

class Cuadrado extends Forma {
    constructor(lado) {
        super();
        this.lado = lado;
    }

    area() {
        return this.lado ** 2;
    }
}

class Triangulo extends Forma {
    constructor(base, altura) {
        super();
        this.base = base;
        this.altura = altura;
```

```
    }

    area() {
        return (this.base * this.altura) / 2;
    }
}

const formas = [
    new Circulo(5),
    new Cuadrado(4),
    new Triangulo(3, 4)
];

formas.forEach((forma) => {
    console.log(`Área: ${forma.area()}`);
});
// Área: 78.53981633974483
// Área: 16
// Área: 6
```

En este ejemplo, cada clase derivada (Circulo, Cuadrado, Triangulo) anula el método area() de la clase base Forma. Cuando iteramos sobre las formas y llamamos al método area() en cada instancia, obtenemos el área correcta según el tipo de forma.

9.2. Polimorfismo a través de interfaces (usando duck typing)

JavaScript no tiene soporte nativo para **interfaces** como otros lenguajes de programación como Java, pero se puede lograr un comportamiento similar utilizando el concepto de "duck typing".

El duck typing se basa en la idea de que si un objeto se comporta como un pato (es decir, tiene las mismas propiedades y métodos), entonces se puede tratar como un pato, independientemente de su tipo real.

Para ilustrar cómo se puede usar duck typing para lograr el **polimorfismo**, supongamos que tenemos varios objetos que representan diferentes tipos de vehículos, cada uno con su propio método `conducir()`. Aunque estos objetos no heredan de una clase base común, todos pueden ser tratados como vehículos debido a que tienen el método `conducir()`:

```
const coche = {
    conducir() {
        console.log('El coche está siendo conducido');
    }
};

const camion = {
    conducir() {
        console.log('El camión está siendo conducido');
```

```
    }
};

const motocicleta = {
    conducir() {
        console.log('La motocicleta está siendo conducida');
    }
};

function realizarViaje(vehiculo) {
    vehiculo.conducir();
}

realizarViaje(coche); // El coche está siendo conducido
realizarViaje(camion); // El camión está siendo conducido
realizarViaje(motocicleta); // La motocicleta está siendo conducida
```

En este ejemplo, la función `realizarViaje()` acepta un objeto vehiculo como parámetro y llama al método conducir() en ese objeto.

Aunque `coche`, `camion` y `motocicleta` no heredan de una clase base común, todos tienen el método `conducir()` y, por lo tanto, pueden ser tratados como vehículos de manera polimórfica.

10. Composición

En algunos casos, en lugar de utilizar la herencia para lograr el **polimorfismo**, se puede usar la **composición**. La composición se refiere al proceso de construir objetos más complejos a partir de objetos más simples al componerlos juntos.

En lugar de heredar propiedades y métodos, un objeto puede contener una referencia a otro objeto y delegar ciertos comportamientos a ese objeto.

Por ejemplo, supongamos que tenemos un objeto Persona que puede tener diferentes roles. En lugar de crear múltiples clases que hereden de Persona para cada rol, podemos utilizar la composición para asignar un objeto Rol a cada instancia de Persona:

```
class Persona {
    constructor(nombre, rol) {
        this.nombre = nombre;
        this.rol = rol;
    }

    realizarTarea() {
        this.rol.realizarTarea(this.nombre);
    }
}

class Rol {
    realizarTarea(nombre) {
```

```
        console.log(`${nombre} está realizando una tarea genérica`);
    }
}

class Desarrollador extends Rol {
    realizarTarea(nombre) {
        console.log(`${nombre} está escribiendo código`);
    }
}

class Gerente extends Rol {
    realizarTarea(nombre) {
        console.log(`${nombre} está gestionando un equipo`);
    }
}

const juan = new Persona('Juan', new Desarrollador());
const ana = new Persona('Ana', new Gerente());

juan.realizarTarea(); // Juan está escribiendo código
ana.realizarTarea(); // Ana está gestionando un equipo
```

11. La clase Date

La clase `Date` en JavaScript es un objeto incorporado que permite trabajar con fechas y horas.

Puedes utilizarla para crear, manipular y obtener información sobre fechas y horas.

Aquí hay una breve descripción de cómo crear y utilizar objetos Date en JavaScript.

11.1. Crear un objeto Date

Puedes crear un objeto Date de varias maneras:

Sin parámetros: crea un objeto Date con la fecha y hora actual.

```
let fechaActual = new Date();
```

Con una cadena de fecha: crea un objeto Date a partir de una cadena de fecha en un formato reconocible.

```
let fechaEspecifica = new Date("2023-04-17");
```

Con valores numéricos: crea un objeto Date con los valores numéricos proporcionados como año, mes, día, hora, minutos, segundos y milisegundos.

```
let fechaConValores = new Date(2023, 3, 17, 12, 30, 0, 0); // Año, Mes (0-11), Día,
Hora, Minutos, Segundos, Milisegundos
```

11.2. Agregar atributos de fecha a una clase y utilizarlos

Aquí hay un ejemplo de una clase que tiene un atributo de fecha y cómo puedes utilizarlo:

```
class Evento {
    constructor(nombre, fecha) {
    this.nombre = nombre;
    this.fecha = new Date(fecha);
    }

    obtenerNombre() {
        return this.nombre;
    }

    obtenerFecha() {
        return this.fecha;
    }

    obtenerDia() {
        return this.fecha.getDate();
    }

    obtenerMes() {
        return this.fecha.getMonth() + 1; // Los meses se cuentan de 0 a 11, así que
sumamos 1.
    }

    obtenerAnio() {
        return this.fecha.getFullYear();
    }
}

let evento1 = new Evento("Reunión", "2023-04-17");
console.log(evento1.obtenerNombre()); // Reunión
console.log(evento1.obtenerFecha()); // 2023-04-17T00:00:00.000Z (dependiendo de la
zona horaria)
console.log(evento1.obtenerDia()); // 17
console.log(evento1.obtenerMes()); // 4
console.log(evento1.obtenerAnio()); // 2023
```

En este ejemplo, hemos creado una clase Evento que tiene un atributo de fecha. También hemos definido varios métodos para obtener información sobre la fecha, como el día, el mes y el año. Estos métodos utilizan los métodos de la clase Date para obtener la información requerida.