DIGITALHOUSE

POO (Programación orientada a objetos)

Programación orientada a objetos

Es un **paradigma** de programación que se basa en el concepto de "**objetos**", los cuales pueden contener *datos*, en forma de ***atributos**, y código*, en forma de *procedimientos*, también conocidos como **métodos**.
En **JavaScript**, un lenguaje de programación orientado a objetos, puedes crear y manipular objetos de manera sencilla.

Clases

En programación orientada a objetos, una clase es una **plantilla** para crear **objetos**. Define un conjunto de **propiedades** y **métodos** comunes que los *objetos* creados *a partir de esa clase* compartirán.

- Una clase es como un "molde" para objetos en la programación.
- Define los atributos (también llamados propiedades o campos) que los objetos tendrán, especificando qué tipo de datos pueden contener.
- También define los métodos, que son funciones asociadas a los objetos de esa clase, y especifica el comportamiento que los objetos pueden realizar.

Constructor

En programación orientada a objetos, un **constructor** es un *método especial* dentro de una **clase** que se llama *automáticamente* cuando se *crea una nueva instancia* de esa clase.

El propósito principal de un **constructor** es *inicializar las propiedades* del objeto y realizar cualquier *configuración inicial* necesaria.

- **Inicialización de atributos**: El constructor se utiliza para asignar valores iniciales a los atributos del objeto.
- **Nombre especial**: El constructor tiene el mismo nombre que la clase. En JavaScript, el constructor se llama constructor.
- **Automáticamente invocado**: No necesitas llamar al constructor explícitamente; se ejecuta automáticamente cuando creas una nueva instancia de la clase utilizando la palabra clave new.

Ejemplo Clase persona

Persona es la **clase** que representa a una persona.

El **constructor** se llama automáticamente cuando se crea una nueva instancia de la clase. Establece los **atributos** *nombre* y *edad* para cada **objeto**.

obtenerInformacion es un **método** de la clase que devuelve una cadena con la información de la persona.

La POO en JavaScript se basa en **prototipos**, y las clases en JavaScript son solo una sintaxis más cómoda para trabajar con prototipos.

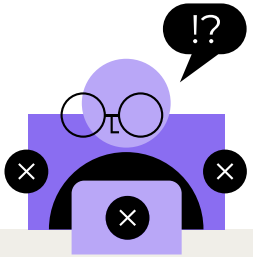
```
// Definición de una clase llamada "Persona"
class Persona {
  // Constructor para inicializar instancias de la clase
  constructor(nombre, edad) {
    this.nombre = nombre;
    this.edad = edad;
  }

  // Método para obtener información de la persona
  obtenerInformacion() {
    return `Nombre: ${this.nombre}, Edad: ${this.edad} años`;
  }
}

// Creación de instancias de la clase Persona
const persona1 = new Persona("Juan", 25);
const persona2 = new Persona("María", 30);

// Uso de métodos para obtener información de las personas
console.log(persona1.obtenerInformacion());
console.log(persona2.obtenerInformacion());
```

¿Qué significa “this”?



La palabra clave **this** se refiere al objeto que está siendo creado. En otras palabras, **this** apunta a la *instancia del objeto* que se está inicializando.

Dentro del **constructor**, **this.nombre** y **this.edad** se utilizan para asignar valores a las propiedades nombre y edad de la instancia de **Persona** que está siendo creada.

El uso de **this** es fundamental en la *programación orientada a objetos* para diferenciar entre las **propiedades** de la *instancia actual* y *variables locales* en el ámbito del **constructor**.

Al utilizar **this**, estás indicando que deseas acceder o modificar las propiedades específicas del objeto que está siendo creado.

Herencia

Permite a una **clase** (llamada clase derivada o subclase) heredar las *propiedades y métodos* de **otra clase** (llamada clase base o superclase). La **herencia** facilita la reutilización del código y la organización jerárquica de las clases.

En este ejemplo, **Perro** hereda de Animal, lo que significa que tiene acceso a la *propiedad* **nombre** y al *método* **hacerSonido** de la clase base. La subclase también puede agregar o modificar funcionalidades según sea necesario.

```
// Clase base (superclase)
class Animal {
  constructor(nombre) {
    this.nombre = nombre;
  }

  hacerSonido() {
    console.log("Haciendo un sonido genérico");
  }
}

// Clase derivada (subclase)
class Perro extends Animal {
  // Puedes agregar propiedades o métodos adicionales aquí

  // Puedes sobrescribir métodos existentes
  hacerSonido() {
    console.log("Guau guau");
  }
}

// Crear una instancia de la subclase
const miPerro = new Perro("Fido");

// Acceder a propiedades y métodos de la superclase
console.log(miPerro.nombre); // Imprime: Fido
miPerro.hacerSonido(); // Imprime: Guau guau
```

Encapsulamiento

El encapsulamiento busca **ocultar** los *detalles internos* de implementación de una clase y **exponer solo la interfaz necesaria** para interactuar con esa **clase**.

Datos privados: Las propiedades de una clase pueden ser declaradas como privadas, lo que significa que solo pueden ser accedidas o modificadas internamente dentro de la propia clase. Esto ayuda a proteger los datos internos de la clase.

Métodos públicos: Los métodos de la clase se definen para interactuar con los datos privados. Estos métodos, a menudo llamados "métodos de acceso" o "getters/setters", proporcionan una interfaz pública para manipular los datos internos de la clase.

```
class CuentaBancaria {
  // Propiedad privada
  #saldo = 0;

  // Método público para obtener el saldo
  obtenerSaldo() {
    return this.#saldo;
  }

  // Método público para depositar dinero
  depositar(cantidad) {
    if (cantidad > 0) {
      this.#saldo += cantidad;
      console.log(`Depósito exitoso. Nuevo saldo: ${this.#saldo}`);
    } else {
      console.log("Error: La cantidad debe ser mayor que cero.");
    }
  }

  // Método público para retirar dinero
  retirar(cantidad) {
    if (cantidad > 0 && cantidad <= this.#saldo) {
      this.#saldo -= cantidad;
      console.log(`Retiro exitoso. Nuevo saldo: ${this.#saldo}`);
    } else {
      console.log("Error: Cantidad inválida o saldo insuficiente.");
    }
  }
}

// Crear una instancia de la clase CuentaBancaria
const miCuenta = new CuentaBancaria();

// Acceder a métodos públicos para interactuar con la cuenta
miCuenta.depositar(1000);
miCuenta.retirar(500);
console.log(`Saldo actual: ${miCuenta.obtenerSaldo()}`);
```

Polimorfismo

Es la capacidad de un **objeto** de tomar varias formas o comportarse de diferentes maneras según el contexto.

En JavaScript, el **polimorfismo** puede expresarse de varias maneras, siendo una de ellas a través de la capacidad de un objeto de utilizar métodos de la **misma firma** (nombre y parámetros) pero con implementaciones diferentes.

En este ejemplo, **Animal** es la clase base y **Perro** y **Gato** son clases derivadas que implementan el método **hacerSonido**. La función **hacerRuido** acepta cualquier objeto que tenga un método **hacerSonido** y lo llama, lo que demuestra el polimorfismo. De esta manera, se pueden tratar objetos de clases diferentes de manera uniforme cuando implementan una interfaz común.

```
// Clase base
class Animal {
  hacerSonido() {
    console.log("Sonido genérico de animal");
  }
}

// Clases derivadas (polimorfismo)
class Perro extends Animal {
  hacerSonido() {
    console.log("Guau guau");
  }
}

class Gato extends Animal {
  hacerSonido() {
    console.log("Miau");
  }
}

// Función que acepta cualquier objeto que tenga el método hacerSonido
function hacerRuido(animal) {
  animal.hacerSonido();
}

// Crear instancias de las clases derivadas
const miPerro = new Perro();
const miGato = new Gato();

// Llamar a la función con diferentes instancias
hacerRuido(miPerro); // Imprime: Guau guau
hacerRuido(miGato);  // Imprime: Miau
```

Prototipo

prototype: Es una propiedad que está presente en todas las **funciones** en **JavaScript**. Cuando creas una función constructora y la utilizas para crear objetos con **new**, el prototipo de esos objetos se establece en la propiedad prototype de la función constructora.

__proto__: Es una propiedad presente en todos los objetos y apunta al prototipo del objeto.

En este ejemplo, **Animal** es una función constructora, y **miAnimal** es un objeto creado a partir de esta función. **miAnimal** hereda el método **hacerSonido** del **prototipo** de **Animal**. Es importante destacar que la relación entre **objetos y prototipos** permite la creación de cadenas de **herencia** en JavaScript.

```
// Función constructora
function Animal(nombre) {
  this.nombre = nombre;
}

// Agregar método al prototipo de Animal
Animal.prototype.hacerSonido = function() {
  console.log("Haciendo un sonido genérico");
};

// Crear un objeto usando la función constructora
const miAnimal = new Animal("Criatura");

// Acceder a propiedad y método
console.log(miAnimal.nombre);      // Imprime: Criatura
miAnimal.hacerSonido();            // Imprime: Haciendo un sonido genérico

// Acceder al prototipo directamente (no recomendado en la práctica)
console.log(miAnimal.__proto__ === Animal.prototype);  // Imprime: true
```