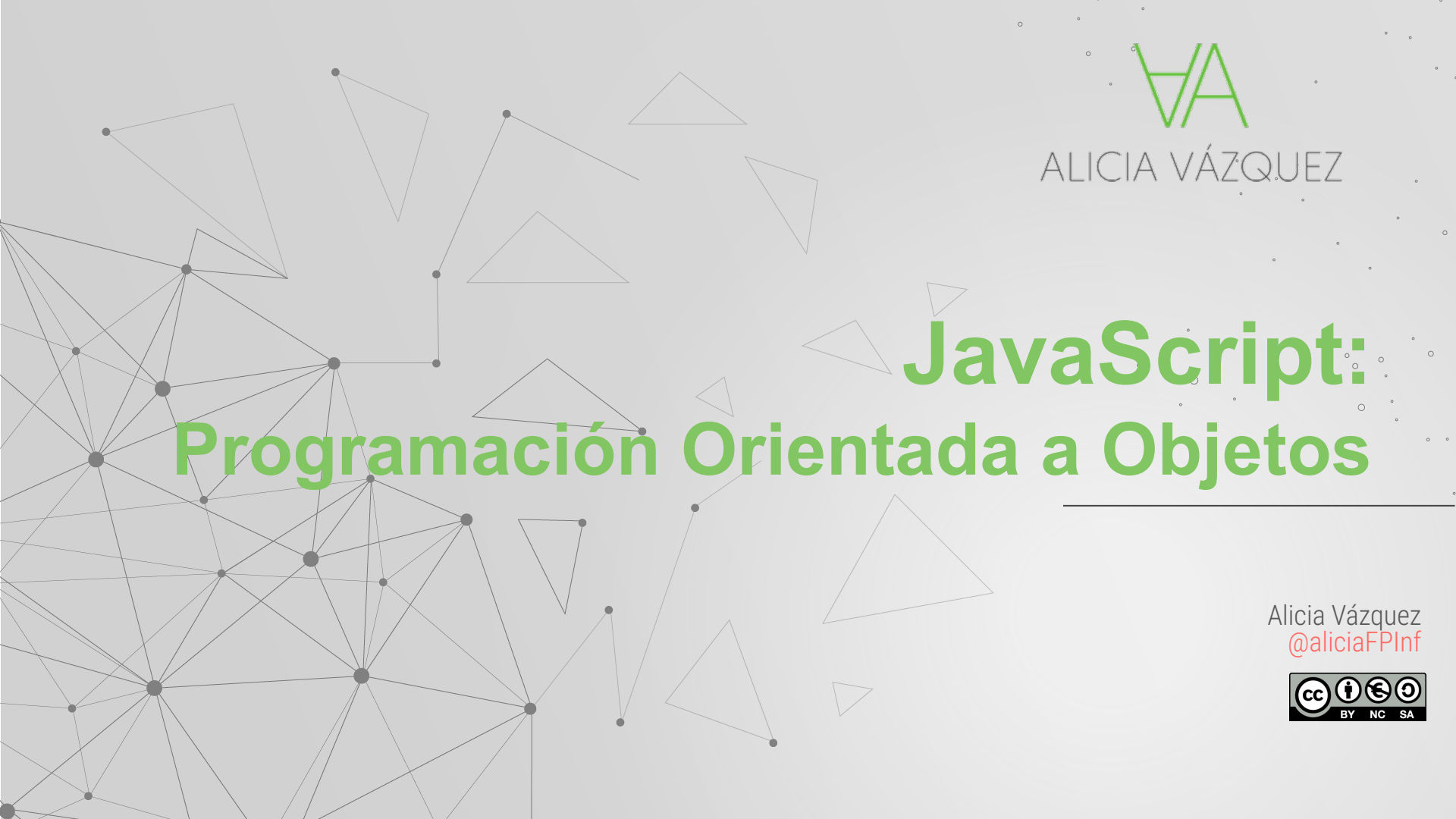


JavaScript:
Programación Orientada a Objetos

Alicia Vázquez
@aliciaFPInf

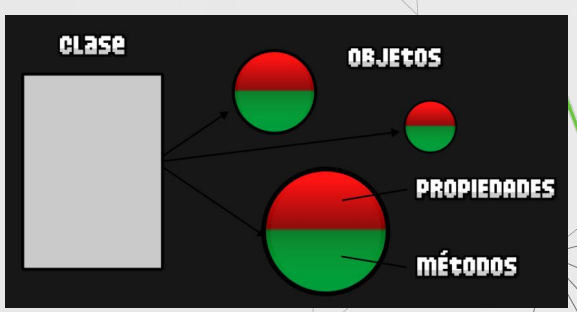
POO

La **Programación Orientada a Objetos** (POO, o en inglés OOP) es un estilo de programación muy utilizado, donde creas y utilizas **estructuras de datos** de una forma muy similar a la "vida real".

En este paradigma de la programación la dificultad radica en definir las diferentes estructuras:

- **Propiedades** → Características de esa estructura.
- **Métodos** → Comportamiento, que puede o no hacer.

A esta estructura la llamaremos **CLASE**. Este es un concepto abstracto, un molde. Pero será gracias a esta definición de esta clase que podremos crear tantos objetos como queramos, siendo cada uno de ellos diferentes.



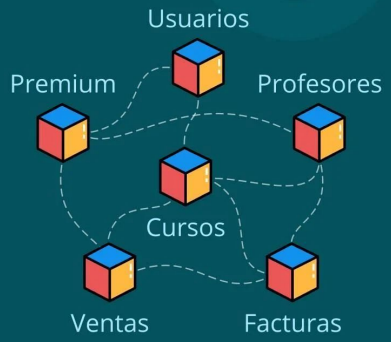
¿QUÉ ES LA PROGRAMACIÓN ORIENTADA A OBJETOS?

Es un **paradigma** de programación que **organiza** las **funciones** en **entidades llamadas objetos**.

Los **objetos se crean** a partir de **una plantilla** llamada **clase**. Cada objeto es una **instancia de su clase**.

Los objetos tienen **datos (atributos)** y **funcionalidades (métodos).**

En una aplicación los objetos están separados **pero se comunican entre ellos.**



ATRIBUTOS

```
Nombres
Apellidos
Correo
Contraseña
Premium
```

MÉTODOS

```
Editar perfil
Iniciar sesión
Cerrar sesión
Cambiar contraseña
Pasar a premium
```

Puedes programar con este paradigma **en la mayoría de lenguajes.**

Aprende a crear aplicaciones usando la POO en:

ed.team/cursos/poo

EDteam

¡PILARES DE LA PROGRAMACIÓN ORIENTADA A OBJETOS!

ABSTRACCIÓN

Es el proceso de **definir los atributos y los métodos** de una clase.

ENCAPSULAMIENTO

Protege la información de manipulaciones no autorizadas.

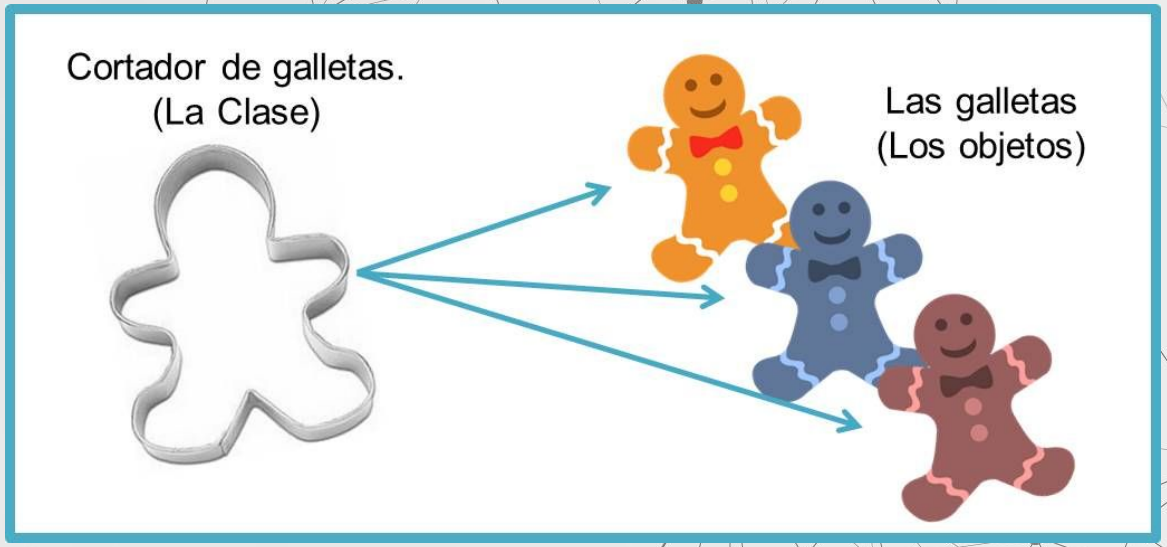
POLIMORFISMO

Da la misma orden a varios objetos para que respondan de diferentes maneras.

HERENCIA

Las **clases hijo heredan atributos y métodos** de las clases padre.

Según el paradigma, la programación orientada objetos, se basa en estos 4 pilares. **Estos definen la simplicidad y la funcionalidad del código.**



01

Clases vs Objeto

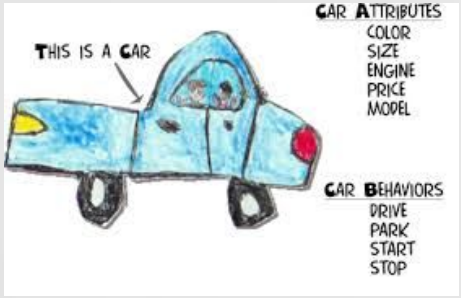
La **clase** es la abstracción el **objeto** es la concreción

CLASE

Definir una clase implica un nivel de **abstracción** puesto que buscamos representar la realidad.

Definiremos una clase usando la palabra **`class`** seguido del nombre de la clase.

Las **propiedades** nos permiten guardar el estado del objeto y los **métodos** son los que definen el comportamiento de todos los objetos de ese tipo.




```
class Coche{
  //Propiedades
  marca;
  modelo;
  matricula;
  potencia;
  precio;
  color;
  constructor() {
      this.marca = "Seat";
      this.modelo = "Ibiza";
      this.color = "blanco";
      this.precio = 21000;
    }
  //Métodos
  informa(){}
  arranca(){}
  para(){}
  acelera(){}
  frena(){}
}
```

CLASE

El **constructor** crea un objeto del tipo de la clase usada y lo deja en un estado válido (con las propiedades que sean necesarias inicializadas). Ya sea con los parámetros recibidos o con los valores por defecto.

Las propiedades que se encuentren inicializadas en el constructor no hace falta que estén declaras como propiedades, pero por simulitud a otros lenguajes o claridad de diseño podemos ponerlas.

En JS solo existe un constructor.

```
class Coche{
   constructor(matricula, potencia) {
//Estas son las propiedades del coche
        this.marca = "Seat";
        this.modelo = "Ibiza";
        this.color = "blanco";
        this.precio = 21000;
        this.matricula = matricula;
        this.potencia = potencia;
      }
   //Métodos
   informa(){}
   arranca(){}
   para(){}
   acelera(){}
   frena(){}
}
```

Constructor

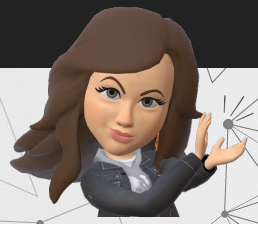
Si lo que realmente queremos es **instanciar** un objeto de una **clase**, usaremos el **constructor**, que no es más que una función que se llama automáticamente al llamar a **new** y que "inicializa" las propiedades a unos valores por defecto o a los valores que le pasemos por parámetro.

En el constructor también podemos definir métodos y guardarlos en una variable, pero no es lo más recomendable puesto que habrá un función para cada instancia.

Dentro del constructor usaremos ***this.*** Para hacer referencia a las propiedades de esta instancia concreta.

```
constructor(marca, modelo, color, potencia) {
        this.marca = marca;
        this.modelo = modelo;
        this.color = color;
        this.precio = 21000;
        this.potencia = potencia;
        this.informa = function() {
            alert( "Coche nuevo: " + this.modelo );
        };
    }
```

```
let miCoche2 = new Coche("Volkswagen",
"Polo","azul",90);
miCoche2.informa();
```



Propiedades Public o Private

Cuando definimos cuáles son los **atributos públicos o privados** de una clase, así como los **métodos que permiten acceder o modificarlos**, estamos aplicando el principio de **encapsulación**.

Una **propiedad pública** es aquella a la que se puede acceder **tanto desde dentro como desde fuera de la clase**, directamente a través de la instancia del objeto.

Una **propiedad privada** es aquella a la que **solo se puede acceder desde el interior de la clase**. Desde fuera no es posible leerla ni modificarla directamente. Si necesitamos acceder a su valor o cambiarlo, debemos hacerlo mediante **métodos específicos**, normalmente **getters y setters**.

Nombre	Sintaxis	Descripción
Propiedad pública	`name` o `this.name`	Se puede acceder a la propiedad desde dentro y fuera de la clase.
Propiedad privada	`#name` o `this.#name`	Se puede acceder a la propiedad sólo desde dentro de la clase.

Propiedades de acceso: Getters y Setters

Cuando definimos un método con la palabra clave **get** o **set** seguido del nombre de la propiedad, el lenguaje nos permite **utilizarlo como si fuera una propiedad y no como una función**.
Esto significa que podemos leer o asignar valores usando el operador **=** sin llamar explícitamente a un método, aunque internamente se esté ejecutando código.

En JavaScript, los **getters** y **setters** permiten definir propiedades de acceso, que se usan como atributos normales pero ejecutan código internamente, facilitando la encapsulación y el cumplimiento de reglas de negocio sin exponer la implementación.

```
class Coche {
 #marca;
 #modelo;
 constructor(marca, modelo) {
   this.#marca = marca;
   this.#modelo = modelo;
 }
 // Getter → se usa como una propiedad
 get informacion() {
   return `${this.#marca} ${this.#modelo}`;
 }
 // Setter → se usa con =
 set informacion(valor) {
   const partes = valor.split(" ");
   if (partes.length !== 2) {
     throw new Error("Formato incorrecto.");
   }
   this.#marca = partes[0];
   this.#modelo = partes[1];
 }
}
```

```
class Coche {
    //Propiedades privadas
    #_matricula;
    //Y las que quiera públicas que no esten en el constructor
    precio = 0;
   constructor(marca, modelo, color, potencia) {
       this.marca = marca;
       this.modelo = modelo;
       this.color = color;
       this.potencia = potencia;
       this.#_matricula = "1234 ABC";
   }
       //Métodos que definen el comportamiento
   informa() {
           console.log( "Coche nuevo: " + this.modelo );
   };
   arranca() { }
   para() { }
   acelera() { }
   frena() { }
   calculaPrecio(){return this.precio * 219}
   //getters y setters que encapsulan
   get matricula(){return this.#_matricula}
   set matricula(matricula){this.#_matricula = matricula;}
}
```

```
const miCoche = new
Coche("seat","ibiza","blanco","110cv");
miCoche.precio = 100;
console.log(`Mi coche es un ${miCoche.modelo}
con matricula ${miCoche.matricula} con precio
${miCoche.calculaPrecio()}`)
```

```
Mi coche es un ibiza con matricula 1234 ABC con precio 21900
```

Métodos

Un **método** es el nombre que recibe una función que existe dentro de una clase. Se utilizan para englobar comportamientos o funcionalidades relacionadas en conjunto con la clase y mediante las cuales podemos segmentar y separar en bloques de código.

Desde un método de la clase podemos acceder a todas las propiedades, ya sean públicas o privadas, pero siempre usando **this.**

```
function informacionCompleta() {
     return "El coche con matrícula: "+this.#matricula +" es un "this..modelo + ' ' + this.marca;
   }
```

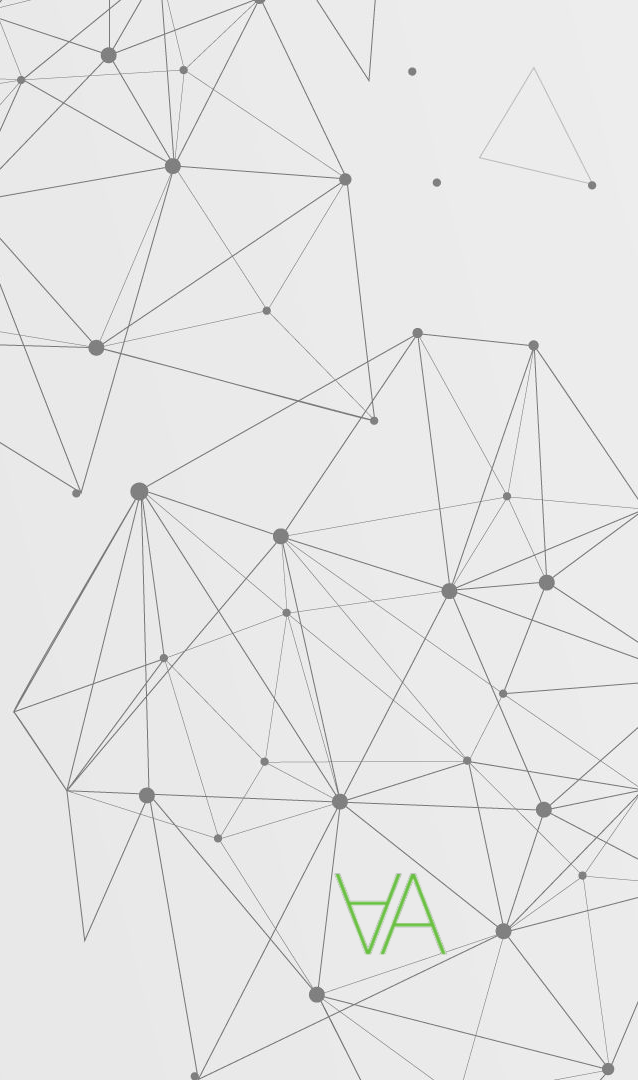
```
El coche con matricula: 1234 ABC es un ibiza seat
```

Ejercicio 1

1. Crea la clase **TarjetaCredito** como abstracción de lo que conoces de una tarjeta de crédito.
2. Decide el nivel de encapsulación, por lo tanto que propiedades son públicas/privadas y que métodos necesitas.
3. Instancia 3 objetos diferentes.
4. Modifica la clase y añade dos o tres métodos que representan el comportamiento de lo que sería una tarjeta.
    - Activar
    - Anular
    - Pagar
    - Cambiar pin

02

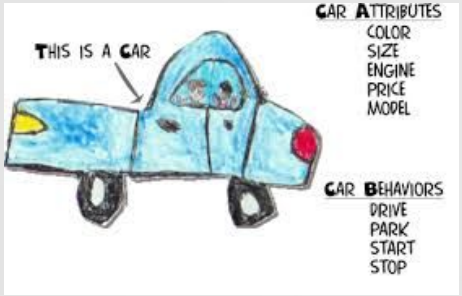
Herencia y polimorfismo



CLASE

Habíamos definido la clase **Coche** anteriormente, con una serie de propiedades y de métodos.

Si nos paramos a pensar, algunos vehículos tienen otras o más propiedades y también pueden tener más comportamientos (métodos). ¿Cómo lo hacemos?



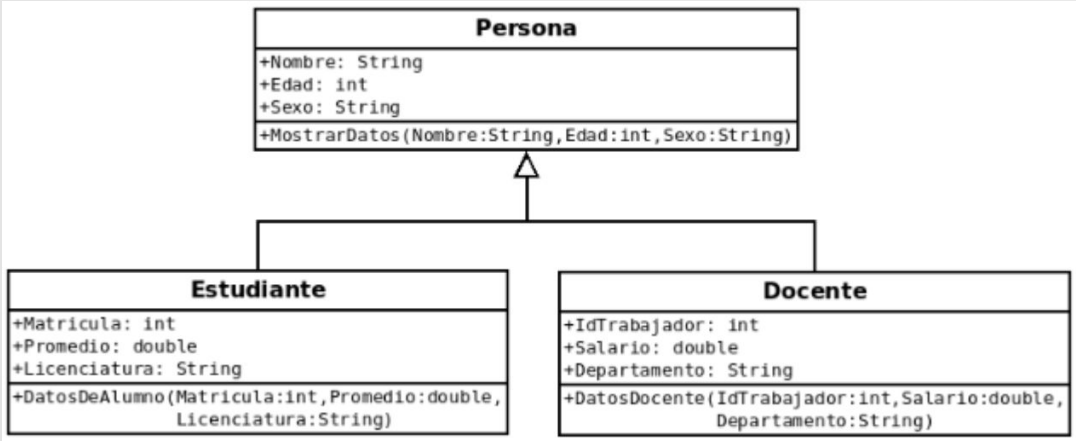

```
class Coche{
  //Propiedades
  marca;
  modelo;
  #_matricula;
  potencia;
  precio;
  color;
  constructor(marca, modelo, color,
potencia) {
       this.marca = "Seat";
       this.modelo = "Ibiza";
       this.color = "blanco";
       this.precio = 21000;
        this.#_matricula = "1234 ABC";
    }
  //Métodos
  informa(){}
  arranca(){}
  para(){}
  acelera(){}
  frena(){}
}
```

Herencia

Una clase puede heredar de otra utilizando la palabra reservada **extends**.
La clase hija hereda TODOS los **métodos y propiedades públicas** de la clase padre.

Si la clase hija define su propio constructor, debe llamar obligatoriamente al constructor de la clase padre mediante **super()** antes de poder acceder a **this**.

Las **propiedades privadas** (definidas con **#**) no se heredan y solo pueden ser utilizadas dentro de la clase en la que fueron declaradas.



Herencia

```
class Autocaravana extends Coche{
  camas;
  constructor(marca, modelo, color, potencia, camas) {
      super(marca, modelo, color, potencia);
      this.camas = camas;
  }
  nivelar(){}
}

let miAutocaravana = new Autocaravana("Volkswagen", "Caravelle","azul",90, 4);
miAutocaravana.matricula = "9876 ZAS"; // No va, es privada. Estamos creando un nuevo atributo para esta instancia.
console.log(`Mi autocaravana es un ${miAutocaravana.modelo} con matricula ${miAutocaravana.getMatricula()}`)
```

Sobreescritura

- Cuando una clase hereda de otra, puede sobreescribir métodos.
- Esto significa que un método definido en la clase padre puede volver a definirse en la clase hija para modificar o ampliar su comportamiento.

  Desde un método sobrescrito en la clase hija, es posible invocar el método original de la clase padre utilizando **super.metodo().**

```
class Autocaravana extends Coche{
  arranca(){
      return "Brum brum";
  }
  para(){
      return super.para() + " Acuerdate de nivelar";
  }
}
console.log(miAutocaravana.arranca());
console.log(miAutocaravana.frena()); // Llama al método de coche directamente.
console.log(miAutocaravana.para());
```

```
Brum brum
Pisa el freno
Quita la llave Acuerdate de nivelar
```

Ejercicio 2

Escribe un programa en JavaScript que modele una jerarquía de figuras geométricas.

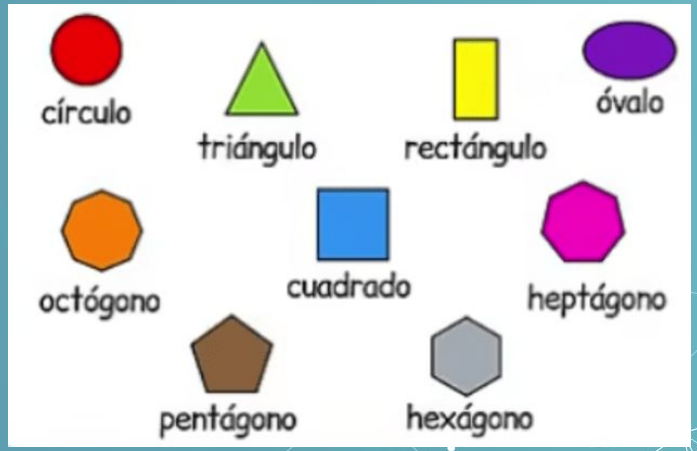
Debe existir una clase base llamada **FiguraGeometrica** que contenga:

- una propiedad **nombre**, que representa el nombre de la figura
- un método **calcularArea(),** que no estará implementado en la clase base y deberá ser redefinido obligatoriamente por las subclases

A continuación, crea al menos tres subclases:

- Rectangulo
- Triangulo
- Circulo

Ambas deben heredar de **FiguraGeometrica** e implementar su propia versión del método **calcularArea**() para calcular correctamente el área de cada figura.



NOTA: JavaScript no dispone de clases ni métodos abstractos de forma nativa.
Para simular este comportamiento, la clase base puede definir el método **calcularArea()** lanzando un error que indique que debe ser implementado por las subclases.

Herencia de Object

Al trabajar con objetos, en muchas ocasiones queremos mostrarlos por pantalla, compararlos, ordenarlos o usarlos en expresiones donde JavaScript espera un valor primitivo.

Para poder hacerlo, JavaScript intenta convertir el objeto a un tipo de dato primitivo (string, number, etc.).
Durante este proceso de conversión, el motor de JavaScript busca y utiliza los métodos:

- toString()
- valueOf()

Todos los objetos, independientemente de su clase o tipo, heredan estos métodos, ya que forman parte de Object.
Sin embargo, podemos sobreescribirlos en nuestras propias clases para definir cómo queremos que el objeto se represente o se comporte durante estas conversiones.

Por defecto:

- El método toString() devuelve la cadena "[object Object]".
- El método valueOf() devuelve el propio objeto.

Al sobrescribir estos métodos, podemos controlar cómo se muestra un objeto o cómo participa en operaciones que requieren valores primitivos.

Object.toString()

Cuando JavaScript necesita convertir un objeto a cadena de texto, intenta llamar primero al método **toString().**

Si este método no existe o no devuelve un valor primitivo, JavaScript intenta entonces llamar a valueOf().

Por este motivo, toString() tiene prioridad en las conversiones a string.

Ejemplo:

- *métodos de ordenación como sort()*
- *concatenaciones con cadenas de texto*

Object.valueOf()

Cuando JavaScript necesita convertir un objeto para operaciones numéricas o comparaciones, intenta llamar primero al método **valueOf().**

Si este método no existe o no devuelve un valor primitivo, se utiliza entonces toString().

Por ello, valueOf() tiene prioridad en conversiones numéricas.

Ejemplos:

- *operadores de comparación (==, !=, <, <=, etc.)*
- *operaciones matemáticas*

Nota aclaratoria

Este proceso de conversión es automático y depende del **contexto de uso del objeto**.
No llamamos explícitamente a estos métodos, sino que JavaScript los utiliza internamente durante el cambio de tipos.

```
class Persona {
    constructor(nombre, edad) {
            this.nombre = nombre;
            this.edad = edad;
    }
    toString() { //Segundo
       return this.nombre;
    }
    valueOf(){//Primero
        return this.edad;
    }
}

let p1 = new Persona("Alicia", 45);
let p2 = new Persona("Zoilo", 15);

console.log("p1>p2?");
console.log(p1 > p2); // Contexto numérico → valueOf()

console.log("p1 " + p1); //Contexto value por operación matemática → valueOf()
console.log(`p1 ${p1}`); // Contexto string → toString()
console.log(p1); // Consola muestra el objeto (no conversión implícita)
```

```
p1>p2?
true
p1 45
p1  Alicia
▶ Persona {nombre: 'Alicia', edad: 45}
```

Interfaces y clases abstractas

JavaScript **no** tiene **clases abstractas**, pero podemos simularlas usando validaciones en el constructor y métodos que lanzan errores, obligando a ser sobreescritos en clases que hereden..

JavaScript **no** tiene **interfaces** como TypeScript o Java, pero podemos simularlas con mixins.

Ejercicio 3

Modela una jerarquía de personajes de un videojuego.

Debe haber una clase base llamada **Personaje** que tenga propiedades comunes a todos los personajes, como *nombre*, *nivel* y *puntosDeVida*. Además, debe tener un método llamado *atacar*() que simule el ataque de un personaje.

Luego, crea al menos dos subclases: **Guerrero** y **Mago**. Ambas subclases deben heredar de Personaje y deben tener propiedades y métodos específicos de cada tipo de personaje. Por ejemplo, un Guerrero podría tener una propiedad fuerza y un método *golpeEspada*(), mientras que un Mago podría tener una propiedad mana y un método *lanzarHechizo*().

Crea varios magos y guerreros y mira de ordenarlos por nivel.

1. Crea una clase **Personaje** con las propiedades:
2. Crea dos **subclases** que hereden de Personaje pero cada una con sus métodos y/o propiedades propias.
3. Crea varios objetos **Guerrero** y **Mago** y guárdalos en un mismo array. Recorre el **array** y llama al método **atacar**() de cada personaje.
4. Ordena el array de personajes por nivel de menor a mayor y muéstralos por consola.
5. Sobrescribe **toString**() y/o **valueOf**() en Personaje para facilitar la visualización y la ordenación de los personajes.

Pista: todos los personajes deben poder tratarse como Personaje, aunque su comportamiento sea distinto.

03

Inyección de dependencias

Inyección de Dependencias

La inyección de dependencias (DI) es un patrón de diseño que permite desacoplar clases al pasarles las dependencias desde el exterior en lugar de instanciarlas dentro de la clase.

Consiste en proporcionar a una clase los objetos que necesita desde el exterior, en lugar de crearlos dentro de la propia clase.

De esta forma, la clase no decide qué dependencias usar, solo sabe **cómo** utilizarlas.

Coche **depende directamente** de `Motor`

No podemos cambiar el motor fácilmente

La clase está **fuertemente acoplada**

Es difícil de probar o reutilizar

Composición: Tiene un...

```
class Motor {
    encender() {
      return "El motor encendido.";
    }
  }
   class Coche {
    constructor() {
      this.motor = new Motor();
    }
     arrancar() {
      return this.motor.encender();
    }
  }
// Crear instancia de Coche
 const coche = new Coche();
 console.log(coche.arrancar());
```

Inyección de Dependencias

Una clase no debería crear lo que necesita, sino recibirlo ya creado.

Coche **no crea el** motor

Solo necesita los métodos que le permitan manejar el motor e inicializar como: **encender()**

Podemos cambiar el motor por otro fácilmente

La clase es más flexible y reutilizable

```
class Motor {
    encender() {
        return "El motor encendido.";
    }
}
class Coche {
    constructor(motor) {
        this.motor = motor;
    }
    arrancar() {
        return this.motor.encender();
    }
}
// Inyección de dependencias
const motor = new Motor();
const coche = new Coche(motor);
console.log(coche.arrancar());
```

ID: Depende de un...

Ejercicio 3 - Armas

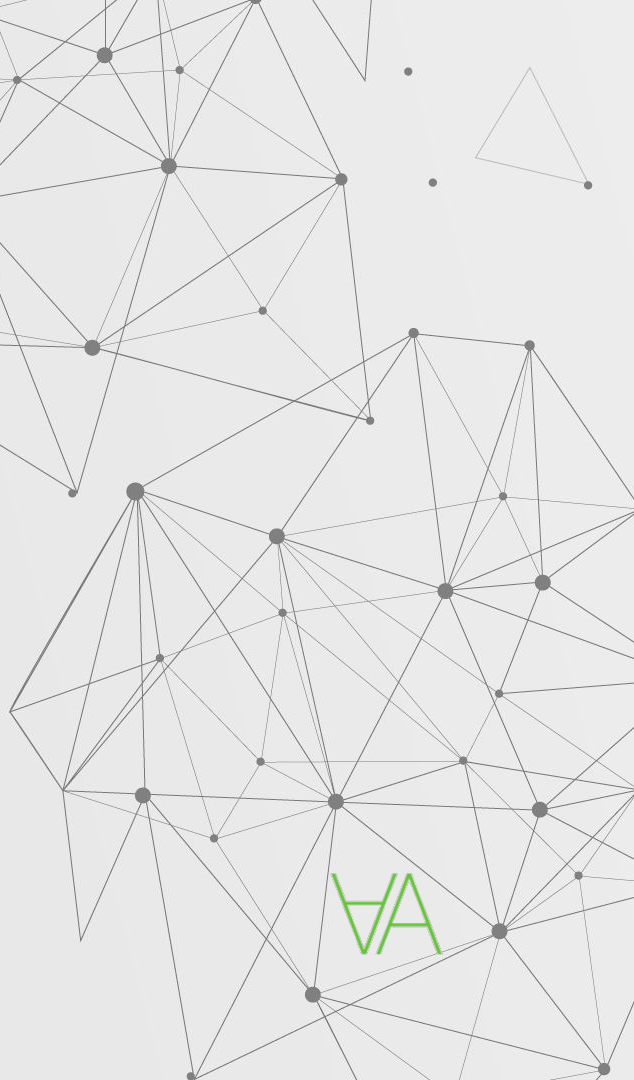
Queremos que armar a nuestros magos y guerreros.

Aunque en JS no es necesario crear una interfaz podemos pensar:: cualquier arma debe tener **nombre** y el método **atacar()** que devuelva el daño.

1. Implementa varias armas: Espada, Hacha, BastonMagico, SinArma (no hace falta que crees ninguna Inerfaz o clase Arma si no quieres)
2. Modifica Personaje para que reciba un arma por el constructor (inyección de dependencias).
3. Implementa equiparArma(arma) para cambiar de arma en tiempo de ejecución.
4. Crea una clase Cofre con una lista de armas “tiradas” y permite que un personaje haga recogerArma(nombre) y se la equipe.
5. Crea varios Guerrero y Mago, haz que recojan armas y luego ordénalos por nivel.

04

Buenas prácticas



Mejor manera de programar en POO con JavaScript

Algunas buenas prácticas para escribir código limpio y escalable en POO con JavaScript:

- Usar **clases** y **encapsulación**: Definir propiedades privadas (#) y exponer solo lo necesario con *get* y *set*.
- Aplicar el principio de **responsabilidad única**: Cada clase debe tener una sola responsabilidad clara (*Single responsability*).
- **Evitar acoplamiento:** Usar inyección de dependencias para mejorar la modularidad.
- **Herencia con moderación:** Prefiere la composición sobre la herencia excesiva.
- **Aplicar SOLID:** Seguir los principios SOLID para mejorar la mantenibilidad.