

Recuerden poner a  
grabar la clase



Clase 17

# Objetos en JavaScript

Diplomatura UNTREF



# Temario

## Objetos en JavaScript:

- funciones constructoras
- propiedades y métodos
- getter y setter
- propiedades privadas
- propiedades sólo lectura
- uso de THIS
- Clases JS

# Objetos en Javascript

# Objetos

En JavaScript, los **objetos** son estructuras de datos fundamentales que se utilizan para representar y almacenar información.

Los **objetos** son colecciones de propiedades, donde cada propiedad tiene un nombre (también llamado clave o identificador) y un valor asociado.



# Objetos

Estos valores pueden ser de cualquier tipo de dato, incluyendo números, cadenas de texto, funciones u otros objetos.

Los **objetos** se utilizan para modelar y organizar datos de una manera más estructurada y flexible que otros tipos de variables simples.



# Objetos: estructura

Un objeto en JavaScript se define utilizando **llaves** `{}` y contiene una **lista** de pares **clave-valor** separados por comas.

**persona** es un objeto que tiene tres propiedades: nombre, edad y ocupacion.

Puedes acceder a las propiedades de un objeto utilizando la notación de punto (**persona.nombre**) o la notación de corchetes (**persona["nombre"]**).

```
let persona = {
  nombre: "Juan",
  edad: 30,
  ocupacion: "Programador"
};
```

# Objetos: Metodos

Los **objetos** en JavaScript también pueden tener **métodos**, que son funciones que están asociadas al objeto y pueden realizar acciones relacionadas con ese objeto.

```
let coche = {  
  marca: "Toyota",  
  modelo: "Camry",  
  arrancar: function() {  
    console.log("El coche está arrancando.");  
  }  
};  
  
coche.arrancar(); // Llama al método "arrancar" del objeto coche.
```

Los objetos son una parte fundamental de JavaScript y se utilizan ampliamente en la programación para representar y manipular datos de manera estructurada y flexible.



# Funciones constructoras

# Funciones constructoras: ¿Que es?

Una **función constructora** es una función especial en JavaScript que se utiliza para crear objetos.

Esta función define la estructura y las propiedades iniciales de los objetos que se crearán a partir de ella.

Los objetos creados con una función constructora comparten la misma estructura y comportamiento, pero pueden tener diferentes valores para sus propiedades.



# Funciones constructoras: como definir las

Para definir una función constructora, debemos:

1 - Crea una función regular y dale un nombre con la primera letra en mayúscula, lo que es una convención para denotar que es una función constructora.

```
function Persona(nombre, edad) {  
    // Propiedades y métodos aquí  
}
```

# Funciones constructoras: como definir las

2 - Dentro de la función **constructora**, puedes definir las propiedades del objeto usando **this**.

**this** se refiere al objeto que se creará cuando llamemos a la función con **new**.

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

# Funciones constructoras: como definir las

Opcionalmente, puedes definir **métodos** dentro de la **función constructora**.

Los **métodos** son funciones que pueden ser llamadas en los objetos creados a partir de la función constructora.

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

# Funciones constructoras: creacion de objetos

Una vez que hayas definido la **función constructora**, puedes **crear objetos** usando el operador **new**.

Cada vez que utilizas **new Persona(...)**, se crea una nueva instancia de **Persona** con sus propias propiedades y métodos.

```
let persona1 = new Persona("Juan", 30);  
let persona2 = new Persona("María", 25);  
  
persona1.saludar(); // Imprime "Hola, mi nombre es Juan y tengo 30 años."  
persona2.saludar(); // Imprime "Hola, mi nombre es María y tengo 25 años."
```

# Funciones constructoras: ventajas

Te **permiten** crear múltiples objetos con la misma estructura y comportamiento de manera eficiente.

Puedes **encapsular** datos y comportamiento relacionado en un solo objeto.  
Facilitan la programación orientada a objetos en JavaScript.



**Operador new**



# Operador new

El operador **new** en JavaScript se utiliza para crear una instancia de un objeto a partir de una función constructora.

Su **función principal** es la de crear un nuevo objeto, asignar ese objeto a la variable que se le indique y establecer el contexto (**this**) dentro de la función constructora para que se refiera a ese nuevo objeto



# Operador new: en accion

En el siguiente ejemplo:

La **función constructora Persona** se define para crear objetos con propiedades nombre y edad.

Luego, se utiliza el **operador new** para llamar a la función constructora Persona.

Cuando usamos **new Persona("Juan", 30)**, se crea un nuevo objeto vacío.

El **contexto (this)** dentro de la función constructora se refiere a ese nuevo objeto.

Las **propiedades nombre y edad** se establecen en el nuevo objeto con los valores proporcionados ("Juan" y 30).

```
function Persona(nombre, edad) {  
  this.nombre = nombre;  
  this.edad = edad;  
}  
  
let persona1 = new Persona("Juan", 30);
```

# Operador new: en accion

En el siguiente ejemplo:

La **función constructora Persona** se define para crear objetos con propiedades nombre y edad.

Luego, se utiliza el **operador new** para llamar a la función constructora Persona.

Cuando usamos **new Persona("Juan", 30)**, se crea un nuevo objeto vacío.

El **contexto (this)** dentro de la función constructora se refiere a ese nuevo objeto.

Las **propiedades nombre y edad** se establecen en el nuevo objeto con los valores proporcionados ("Juan" y 30).

```
function Persona(nombre, edad) {  
  this.nombre = nombre;  
  this.edad = edad;  
}  
  
let persona1 = new Persona("Juan", 30);
```

# Propiedades y metodos

# Propiedades y metodos: Propiedades

Las **propiedades de un objeto** son como **variables** que almacenan valores.

Estos **valores** pueden ser de cualquier tipo de dato, incluyendo números, cadenas de texto, booleanos, funciones u otros objetos.

Las **propiedades de un objeto** se **definen** utilizando pares clave-valor, donde la clave (también conocida como nombre o identificador) es una cadena de texto que actúa como un nombre para la propiedad, y el valor es el dato que se almacena en esa propiedad.

```
let persona = {  
  nombre: "Juan",  
  edad: 30,  
  soltero: true  
};
```

# Propiedades y metodos: Metodos

Los **métodos** son **funciones** que están asociadas a un objeto.

En otras palabras, son propiedades de un objeto que contienen funciones.

Los **métodos** se utilizan para definir el **comportamiento o las acciones** que un objeto puede llevar a cabo.

Por ejemplo, podrías tener un objeto coche con un método llamado arrancar que hace que el coche comience a funcionar.

```
let coche = {  
  marca: "Toyota",  
  modelo: "Camry",  
  arrancar: function() {  
    console.log("El coche está arrancando.");  
  }  
};
```

# Propiedades y métodos: conclusión

En resumen, las propiedades son como variables que almacenan datos en un objeto, mientras que los métodos son funciones que describen las acciones que un objeto puede realizar.

Estas propiedades y métodos permiten modelar y trabajar con objetos de una manera organizada y eficiente en JavaScript.



getter y setter



# getter y setter

Los **getters** y **setters** son métodos especiales en JavaScript que se utilizan para obtener y establecer el valor de propiedades de un objeto.

Permiten un mayor control y personalización del acceso a las propiedades de un objeto.



# getter y setter

Esto es útil para asegurarse de que los valores asignados o recuperados cumplan con ciertas reglas o validaciones, o para realizar tareas adicionales cuando se accede a una propiedad.



# getter

Un **getter** es un método que se utiliza para obtener el valor de una propiedad específica de un objeto. Se define utilizando la palabra clave `get` seguida del nombre del getter.

```
let persona = {  
  nombre: "Juan",  
  apellido: "Pérez",  
  get nombreCompleto() {  
    return this.nombre + " " + this.apellido;  
  }  
};  
  
console.log(persona.nombreCompleto); // Devuelve "Juan Pérez"
```

# getter en acción

Te explicamos el ejemplo:

**nombreCompleto** es un **getter** que devuelve la concatenación del nombre y el apellido.

Para acceder al valor de la propiedad **nombreCompleto**, simplemente la tratamos como si fuera una propiedad normal, sin necesidad de paréntesis.

```
let persona = {  
  nombre: "Juan",  
  apellido: "Pérez",  
  get nombreCompleto() {  
    return this.nombre + " " + this.apellido;  
  }  
};  
  
console.log(persona.nombreCompleto); // Devuelve "Juan Pérez"
```

# setter

Un **setter** es un método que se utiliza para establecer el valor de una propiedad específica de un objeto.

Se define utilizando la palabra clave `set` seguida del nombre del setter.

```
let persona = {  
  nombre: "Juan",  
  apellido: "Pérez",  
  set nombreCompleto(valor) {  
    let partes = valor.split(" ");  
    this.nombre = partes[0];  
    this.apellido = partes[1];  
  }  
};  
  
persona.nombreCompleto = "María González";  
console.log(persona.nombre); // Devuelve "María"  
console.log(persona.apellido); // Devuelve "González"
```

# setter en acción

Te explicamos el ejemplo:

**nombreCompleto** es un **setter** que recibe un valor como argumento y luego divide ese valor en nombre y apellido para establecer las propiedades correspondientes nombre y apellido.

Para asignar un valor al **setter**, simplemente lo asignamos como si fuera una propiedad normal.

```
let persona = {
  nombre: "Juan",
  apellido: "Pérez",
  set nombreCompleto(valor) {
    let partes = valor.split(" ");
    this.nombre = partes[0];
    this.apellido = partes[1];
  }
};

persona.nombreCompleto = "María González";
console.log(persona.nombre); // Devuelve "María"
console.log(persona.apellido); // Devuelve "González"
```

# getter y setter: ventajas

Entre las ventajas de usarlos tenemos:

## **Control:**

Los getters y setters permiten controlar cómo se accede y modifica una propiedad, lo que puede ser útil para aplicar reglas de validación o lógica adicional.

## **Encapsulación:**

Ayudan a encapsular los datos y comportamientos relacionados en un objeto, lo que es un principio clave de la programación orientada a objetos.

## **Flexibilidad:**

Permiten cambiar la implementación interna de una propiedad sin afectar el código que la utiliza, ya que la interfaz (getter y setter) permanece constante.



propiedades privadas



# Propiedad privada: ¿Que es?

Las propiedades privadas en JavaScript son variables o miembros de una clase que están diseñados para ser inaccesibles y no modificables desde fuera de la propia clase en la que se definen.

Estas propiedades se utilizan para ocultar los detalles internos de una clase y encapsular su comportamiento, lo que es un principio fundamental de la programación orientada a objetos (POO).



# Propiedad privada: ¿Que es?

Para declarar una propiedad como privada en JavaScript, se utiliza la convención o la notación con el carácter # antes del nombre de la propiedad.

A partir de ECMAScript 2020, esta notación se convierte automáticamente en una propiedad privada.

Las propiedades privadas solo pueden ser accedidas y modificadas desde métodos internos de la clase en la que se definen.

```
class MiClase {
  #propiedadPrivada;

  constructor() {
    this.#propiedadPrivada = 42;
  }

  obtenerPropiedadPrivada() {
    return this.#propiedadPrivada;
  }

  modificarPropiedadPrivada(nuevoValor) {
    this.#propiedadPrivada = nuevoValor;
  }
}

const instancia = new MiClase();
console.log(instancia.obtenerPropiedadPrivada()); // 42
instancia.modificarPropiedadPrivada(99);
console.log(instancia.obtenerPropiedadPrivada()); // 99

// Intentar acceder directamente a la propiedad privada daría un error
console.log(instancia.#propiedadPrivada); // Error
```

# Propiedad privada: características

## Encapsulación:

Las propiedades privadas permiten encapsular el comportamiento y los datos de una clase, lo que significa que los detalles internos están ocultos y solo se exponen las interfaces públicas necesarias para interactuar con la clase.

## Seguridad:

Al restringir el acceso a ciertas propiedades, se evita que código externo no autorizado modifique o acceda a datos sensibles o críticos de una clase.



# Propiedad privada: características

## Privacidad:

Las propiedades privadas se marcan con el carácter # antes de su nombre. Por ejemplo, `#nombrePrivado`.

## Acceso Restringido:

Solo los métodos internos de la clase pueden acceder y modificar estas propiedades privadas.



# Propiedad privada: ventajas

## **Mayor Seguridad:**

Las propiedades privadas ayudan a proteger los datos y el comportamiento de una clase al evitar cambios accidentales o maliciosos desde fuera de la clase.

## **Abstracción:**

Permiten a los programadores centrarse en las interfaces públicas de la clase sin preocuparse por los detalles internos.

## **Mantenimiento más Sencillo:**

Al cambiar la implementación interna de una clase con propiedades privadas, no se afecta el código externo que utiliza la clase, siempre que la interfaz pública siga siendo la misma.



# Propiedad privada: ejemplo

la propiedad **#saldo** es privada y solo se puede acceder y modificar a través de los métodos depositar, retirar, y consultarSaldo.

Esto garantiza que el saldo de la cuenta bancaria se maneje de manera segura y que no se pueda acceder directamente desde fuera de la clase.

```
class CuentaBancaria {
  #saldo = 0;

  constructor(nombre, saldoInicial) {
    this.nombre = nombre;
    this.#saldo = saldoInicial;
  }

  depositar(cantidad) {
    if (cantidad > 0) {
      this.#saldo += cantidad;
      console.log(`Depósito de ${cantidad} realizado. Nuevo saldo: ${this.#saldo}`);
    }
  }

  retirar(cantidad) {
    if (cantidad > 0 && cantidad <= this.#saldo) {
      this.#saldo -= cantidad;
      console.log(`Retiro de ${cantidad} realizado. Nuevo saldo: ${this.#saldo}`);
    } else {
      console.log("Fondos insuficientes.");
    }
  }

  consultarSaldo() {
    console.log(`Saldo actual: ${this.#saldo}`);
  }
}

const cuenta = new CuentaBancaria("Juan Pérez", 1000);

cuenta.consultarSaldo(); // Saldo actual: 1000
cuenta.depositar(500);   // Depósito de 500 realizado. Nuevo saldo: 1500
cuenta.retirar(200);     // Retiro de 200 realizado. Nuevo saldo: 1300

// Intentar acceder directamente a la propiedad privada daría un error
console.log(cuenta.#saldo); // Error
```

propiedades Solo lectura

# Propiedad solo lectura: ¿Que es?

Las propiedades **solo lectura** en JavaScript son propiedades de un objeto que solo pueden ser leídas y no pueden ser modificadas después de su inicialización.

Esto significa que una vez que se establece un valor para una propiedad solo lectura, no se puede cambiar ni sobrescribir.





# Propiedad solo lectura: creación

Para crear propiedades solo lectura en JavaScript, puedes utilizar dos enfoques principales:

**Usando `Object.defineProperty`:** es un método en JavaScript que se utiliza para definir o modificar las propiedades de un objeto de manera más detallada y controlada.

Permite configurar una amplia variedad de características de una propiedad, como su valor, si es o no escribible, si es enumerable y si es configurable.

Este enfoque es útil cuando deseas tener un mayor control sobre la propiedad y sus características.

```
const miObjeto = {};

Object.defineProperty(miObjeto,
  'propiedadSoloLectura', {
    value: 'Este es un valor solo lectura',
    writable: false, // Establece la propiedad
                     // como no modificable
  });

console.log(miObjeto.propiedadSoloLectura);
// "Este es un valor solo lectura"
miObjeto.propiedadSoloLectura = 'Nuevo
valor'; // No se permitirá la modificación
console.log(miObjeto.propiedadSoloLectura);
// "Este es un valor solo lectura"
```

# Propiedad solo lectura: creación

**Usando clases con getters:** Puedes definir propiedades como solo lectura utilizando métodos getter en una clase. Los métodos getter permiten acceder a una propiedad como si fuera una propiedad directa del objeto, pero en realidad, el valor se calcula cada vez que se accede a la propiedad.

```
class MiClase {
  constructor() {
    this._propiedadSoloLectura = 'Este es un
valor solo lectura';
  }

  get propiedadSoloLectura() {
    return this._propiedadSoloLectura;
  }
}

const instancia = new MiClase();

console.log(instancia.propiedadSoloLectura);
// "Este es un valor solo lectura"
instancia.propiedadSoloLectura = 'Nuevo
valor'; // No se permitirá la modificación
console.log(instancia.propiedadSoloLectura);
// "Este es un valor solo lectura"
```

# Ventajas de propiedades solo lectura

## **Inmutabilidad:**

Ayudan a mantener la inmutabilidad de ciertos datos en objetos, lo que puede ser útil para garantizar que ciertos valores críticos no sean modificados accidentalmente.

## **Seguridad:**

Proporcionan un nivel adicional de seguridad al evitar que se realicen cambios no autorizados en propiedades importantes de un objeto.

## **Claridad y Control:**

Ayudan a documentar y comunicar claramente que ciertas propiedades no deben cambiar después de la inicialización, lo que facilita el entendimiento y el mantenimiento del código.



# Propiedades solo lectura: conclusión

Las propiedades solo lectura en JavaScript son útiles para garantizar la inmutabilidad y la seguridad de ciertos datos en un objeto, lo que puede mejorar la claridad y la robustez del código.

Puedes elegir el enfoque que mejor se adapte a tus necesidades, ya sea utilizando `Object.defineProperty` o métodos `getter` en clases.



this

# this: ¿Que es?

**this** es una **palabra clave** en JavaScript que se utiliza en el contexto de objetos para hacer referencia al propio objeto en el que se encuentra.

Su valor depende del contexto en el que se utiliza y varía según cómo se invoca una función o método.



# this: ¿Que es?

El uso más común de **this** en objetos es dentro de métodos de objeto.

Cuando una **función** se llama como un método de un objeto, **this** se refiere al objeto en el que se encuentra el método.



## this en acción

En este ejemplo, `this` dentro del método `saludar` se refiere al objeto `persona`, lo que permite acceder a sus propiedades `nombre` y `edad`.

```
const persona = {  
  nombre: 'Juan',  
  edad: 30,  
  saludar: function() {  
    console.log(`Hola, mi nombre es  
    ${this.nombre} y tengo ${this.edad}  
    años.`);  
  }  
};  
  
persona.saludar(); // "Hola, mi nombre es  
Juan y tengo 30 años."
```



## this en acción

En este ejemplo, **this** dentro del método saludar se refiere al **objeto** persona, lo que permite acceder a sus **propiedades** nombre y edad.

Es importante tener en cuenta que el valor de **this** puede cambiar en diferentes contextos.

Por ejemplo, cuando se pasa un método como una función independiente o cuando se utiliza en funciones de flecha.

```
const persona = {  
  nombre: 'Juan',  
  edad: 30,  
  saludar: function() {  
    console.log(`Hola, mi nombre es  
    ${this.nombre} y tengo ${this.edad}  
    años.`);  
  }  
};  
  
persona.saludar(); // "Hola, mi nombre es  
Juan y tengo 30 años."
```

## this en acción

El comportamiento de this en JavaScript es un tema importante y puede ser un poco complejo en ciertos casos.



# Valor de 'this' en funciones normales

En una función normal (no una función de flecha), el valor de `this` depende de cómo se llama la función en ese momento.

Si se llama como método de un objeto, **this** se refiere a ese objeto.

Si se llama como una función independiente, **this** puede variar según el contexto de ejecución.

Puedes utilizar métodos como `call()` o `apply()` para establecer explícitamente el valor de `this` al llamar a una función.

```
function saludar() {
  console.log('Hola, mi nombre es
  ${this.nombre}');
}

const persona = {
  nombre: 'Juan',
  saludar: saludar
};

persona.saludar(); // "Hola, mi nombre es
Juan"

const otraPersona = {
  nombre: 'María'
};

saludar.call(otraPersona); // "Hola, mi
nombre es María"
```

# Valor de 'this' en eventos

En el contexto de eventos en el navegador, como los **manejadores** de eventos **click**, el valor de **this** suele hacer referencia al elemento DOM que desencadenó el evento.

```
const boton =  
document.querySelector('#miBoton');  
  
boton.addEventListener('click', function()  
{  
  console.log(`Hiciste clic en el botón:  
    ${this.textContent}`);  
});
```

# Valor de 'this' en funciones constructor

Cuando se utiliza una función como constructor (mediante el **operador new**), **this** hace referencia al nuevo objeto que se está creando.

```
function Persona(nombre) {  
  this.nombre = nombre;  
}  
  
const juan = new Persona('Juan');  
console.log(juan.nombre); // "Juan"
```

# Valor de 'this' en funciones flecha

las funciones de flecha no tienen su propio valor this. En su lugar, heredan el valor de this del contexto en el que se definen.

```
const objeto = {  
  mensaje: 'Hola',  
  saludar: () => {  
    console.log(this.mensaje); // 'Hola'  
  }  
};  
  
objeto.saludar();
```

# Ojo con this, en funciones anidadas

En funciones anidadas dentro de métodos, el valor de **this** puede cambiar.

En tales casos, es común asignar **this** a otra variable (self o **\_this**) para mantener el valor correcto de **this** en el ámbito interno.

```
const persona = {
  nombre: 'Juan',
  hablar: function() {
    const self = this;
    setTimeout(function() {
      console.log(`Hablo como
${self.nombre}`);
    }, 1000);
  }
};

persona.hablar(); // "Hablo como Juan"
después de 1 segundo
```

# funciones controladoras

**bind()**, **call()** y **apply()**: Estas son funciones que puedes usar para controlar explícitamente el valor de **this** en una función. **bind()** devuelve una nueva función con **this** vinculado a un valor específico, mientras que **call()** y **apply()** permiten llamar a una función con un valor de **this** particular.

```
const persona = {
  nombre: 'Juan',
  hablar: function() {
    const self = this;
    setTimeout(function() {
      console.log(`Hablo como
${self.nombre}`);
    }, 1000);
  }
};

persona.hablar(); // "Hablo como Juan"
después de 1 segundo
```



# funciones controladoras

## **bind():**

Es un método que se utiliza para crear una nueva función que tiene un valor de `this` específico, vinculado a ella. Esto es útil cuando deseas mantener un contexto específico para una función y luego llamarla más tarde.

**nuevaFuncion:** La nueva función que se crea con `bind()` y que tiene `this` vinculado al contexto proporcionado.

**funcion:** La función original a la que deseas vincular `this`.

**contexto:** El valor que deseas que sea `this` cuando se llame a la nueva función.

```
nuevaFuncion = funcion.bind(contexto);
```

# funciones controladoras

## bind():

funcionVinculada se crea utilizando bind() y objeto como contexto, lo que garantiza que this dentro de funcionVinculada se refiera al objeto objeto.

```
const objeto = {  
  x: 10,  
  imprimirX: function() {  
    console.log(this.x);  
  }  
};  
  
const funcionVinculada =  
objeto.imprimirX.bind(objeto);  
funcionVinculada(); // Imprimirá 10
```

# funciones controladoras

## Call():

es un método que permite llamar a una función con un valor de `this` específico, seguido de los argumentos que se pasan como una lista separada por comas.

La sintaxis es la siguiente:

**funcion:** La función que deseas llamar.

**contexto:** El valor que deseas que sea `this` cuando se llame a la función.

**arg1, arg2, ...:** Los argumentos que deseas pasar a la función.

```
funcion.call(contexto, arg1, arg2, ...);
```

# funciones controladoras

## Call():

En el siguiente ejemplo, `call()` se utiliza para llamar a la función `saludar` con `this` vinculado al objeto `persona`.

```
function saludar() {  
  console.log(`Hola, mi nombre es  
    ${this.nombre}`);  
}  
  
const persona = { nombre: 'Juan' };  
  
saludar.call(persona); // Imprimirá "Hola,  
mi nombre es Juan"
```

# funciones controladoras

## **apply()**

es similar a `call()`, pero en lugar de aceptar una lista separada por comas de argumentos, acepta un array de argumentos.

La sintaxis es la siguiente:

**funcion:** La función que deseas llamar.

**contexto:** El valor que deseas que sea `this` cuando se llame a la función.

**[arg1, arg2, ...]:** Un array que contiene los argumentos que deseas pasar a la función.

```
funcion.apply(contexto, [arg1, arg2, ...]);
```

# Funciones controladoras

## apply()

En el siguiente ejemplo, `apply()` se utiliza para llamar a la función `sumar` con `this` vinculado al objeto `objeto` y se pasan los argumentos 2 y 3 como un array.

```
function sumar(a, b) {  
  return this.x + a + b;  
}  
  
const objeto = { x: 10 };  
  
const resultado = sumar.apply(objeto, [2,  
3]); // Resultado será 15
```

## Funciones controladoras: conclusión

Tanto **call()** como **apply()** son útiles cuando necesitas llamar a una función con un contexto específico y proporcionarle argumentos de manera dinámica.

En resumen, **bind()**, **call()**, y **apply()** son métodos que te permiten controlar el valor de **this** en una función y pueden ser útiles en situaciones donde necesitas gestionar el contexto y los argumentos de una función de manera precisa.



classes JS



# Clases JS

En JavaScript, las clases son una forma de definir objetos y su comportamiento utilizando una sintaxis más orientada a objetos y estructurada.

Las clases se introdujeron en ECMAScript 6 (también conocido como ES6 o ECMAScript 2015) para proporcionar una manera más clara y organizada de crear objetos y definir su estructura.



# Clases JS

Una **clase** en JavaScript es como un plano o un prototipo para crear objetos.

Define las propiedades y los métodos que los objetos creados a partir de ella tendrán.

Las clases proporcionan una manera más fácil de crear objetos similares con la misma estructura y comportamiento.



# Clases JS, en acción

Analizaremos el paso a paso de una clase

## Constructor:

El constructor es un método especial dentro de una clase que se llama automáticamente cuando se crea un objeto a partir de la clase. En el ejemplo, el constructor de la clase Persona acepta dos parámetros, nombre y edad, y se utiliza para inicializar las propiedades del objeto creado. Es donde se configuran las características iniciales del objeto.

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

# Clases JS, en acción

Analizaremos el paso a paso de una clase

## Métodos:

Los métodos son funciones que están asociadas a una clase y que pueden ser llamadas en los objetos creados a partir de esa clase. En el ejemplo, saludar es un método de la clase Persona que imprime un mensaje. Los métodos son comportamientos que los objetos pueden realizar.

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

  // Método para saludar
  saludar() {
    console.log(`Hola, soy ${this.nombre} y
tengo ${this.edad} años.`);
  }
}
```

# Clases JS, en acción

Analizaremos el paso a paso de una clase

Instanciación:

La creación de objetos a partir de una clase se llama instanciación.

En el ejemplo, `const persona1 = new Persona("Juan", 30)` crea una instancia de la clase `Persona` llamada `persona1`.

Cada instancia es un objeto independiente con sus propias propiedades y métodos, pero comparten la misma estructura definida por la clase.

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

  // Método para saludar
  saludar() {
    console.log(`Hola, soy ${this.nombre} y
    tengo ${this.edad} años.`);
  }
}

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

# Clases JS, en acción

## Palabra clave this:

La palabra clave `this` se utiliza dentro de los métodos de una clase para hacer referencia a la instancia actual del objeto.

En el constructor de `Persona`, `this.nombre` y `this.edad` se utilizan para asignar los valores proporcionados como parámetros a las propiedades del objeto.

## Encapsulación:

Aunque JavaScript no tiene modificadores de acceso como algunos otros lenguajes de programación (como `"public"` o `"private"`), las clases permiten definir propiedades y métodos como públicos o privados.

Las propiedades y métodos públicos son accesibles desde fuera de la clase, mientras que los privados se utilizan dentro de la clase y no deben ser accedidos directamente desde fuera. La introducción de propiedades privadas mediante el uso del carácter `#` es una adición más reciente al lenguaje para lograr una encapsulación más estricta.

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

  // Método para saludar
  saludar() {
    console.log(`Hola, soy ${this.nombre} y
    tengo ${this.edad} años.`);
  }
}

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

// Llamada a los métodos de las instancias
persona1.saludar(); // Salida: Hola, soy
Juan y tengo 30 años.
persona2.saludar(); // Salida: Hola, soy
María y tengo 25 años.
("María", 25);
```

¿Dudas o consultas?





**¡Muchas Gracias!**