Primera parte - Las bases

1. Introducción a los Lenguajes de Programación

Un **lenguaje de programación** es un conjunto de reglas y símbolos que los programadores utilizan para comunicarse con las computadoras. A través de un lenguaje de programación, puedes dar instrucciones a la computadora para que realice tareas específicas, como cálculos, almacenamiento de datos y control de dispositivos.

JavaScript es un lenguaje de programación que se utiliza para crear aplicaciones interactivas y dinámicas. A continuación, aprenderemos los conceptos básicos de JavaScript.

Variables

Las **variables** son como cajas donde puedes almacenar información. Puedes nombrar estas cajas y poner datos dentro de ellas para usarlos más tarde.

Cómo declarar variables:

- `var`: Una forma antigua de declarar variables. Es mutable. No se recomienda su uso.
- `let`: Una forma moderna que permite declarar variables que son mutables.
- `const`: Se usa para declarar variables inmutables.

Ejemplo:

```
var nombre = "Juan"; // Usando var
let edad = 25; // Usando let
const pais = "España"; // Usando const
```

Tipos de datos

Un **tipo de dato** es una categoría que define la naturaleza de un valor que puede ser almacenado en una variable. En programación, cada variable tiene un tipo de dato asociado, y este tipo determina qué tipo de operaciones se pueden realizar con la variable y cómo se almacena la información en la memoria.

Tipo de Dato	Descripción	Ejemplo
Primitivos		
String	Representa una secuencia de caracteres.	`let nombre = "Juan";`
Number	Representa números, tanto enteros como decimales.	`let edad = 30;`
BigInt	Representa números enteros muy grandes que no pueden ser representados por Number.	`let grande = 12345678901234567890123456 7890n;`
Boolean	Representa un valor lógico: true o false.	`let esMayor = true;`
Undefined	Indica que una variable no ha sido asignada o no tiene un valor definido.	`let x; (x es undefined)`
Null	Representa la ausencia intencional de un valor.	`let vacio = null;`

Operadores

Los operadores son elementos fundamentales en la programación que permiten realizar diversas acciones sobre los datos. En términos simples, un operador toma uno o más valores, los procesa de alguna manera y produce un resultado.

Los operadores son utilizados para manipular variables y realizar cálculos, comparaciones, concatenaciones y otras operaciones. En el contexto de un lenguaje de programación, actúan como instrucciones que indican cómo debe tratarse la información.

Tipo de Operador	Operador	Descripción	Ejemplo
Aritméticos			
	`+`	Suma.	`5 + 3 → 8`
	`-`	Resta.	`5 - 3 → 2`
	`*`	Multiplicación.	`5 * 3 → 15`
	`/`	División.	`6 / 2 → 3`
	`%`	Módulo (resto de la división).	`5 % 2 → 1`
	`**`	Potenciación (exponenciación).	`2 ** 3 → 8`
Asignación			
	`=`	Asignación.	`x = 5`
	`+=`	Suma y asigna.	`x += 2` (equivale a `x = x + 2`)
	`-=`	Resta y asigna.	`x -= 2`
	`*=`	Multiplica y asigna.	`x *= 2`
	`/=`	Divide y asigna.	`x /= 2`
	`%=`	Módulo y asigna.	`x %= 2`
	`**=`	Potencia y asigna.	`x **= 2`

Comparación			
	==	Igualdad (no estricto). Compara el valor, mas no el tipo de dato.	5 == '5' → true
	===	Igualdad estricta. Compara el valor y el tipo de dato.	5 === '5' → false
	!=	Desigualdad (no estricto).	5 != '5' → false
	!==	Desigualdad estricta.	5 !== '5'→ true
	>	Mayor que.	5 > 4 → true
	<	Menor que.	5 < 4 → false
	>=	Mayor o igual que.	5 >= 5 → true
	<=	Menor o igual que.	5 <= 5 → true
Lógicos			
	&&	Y lógico.	true && false → false
	!	Negación lógica.	!true → false
Unary			
	++	Incremento.	x++ (Si X era 4, ahora es 5)
	--	Decremento.	x--
	+	Convierte a número.	+'5' → 5
	-	Convierte a número negativo.	-5 → -5

2. Condicionales

Las **condicionales** te permiten tomar decisiones en tu código. Esto significa que puedes ejecutar diferentes partes del código dependiendo de si algo es verdadero o falso.

Estructura if-elseif-else:

```
let puntuacion = 85;

//Con “console.log()” imprimimos el contenido por consola.

if (puntuacion >= 90) {
    console.log("Excelente");
} else if (puntuacion >= 75) {
    console.log("Bien");
} else if (puntuacion >= 60) {
    console.log("Suficiente");
} else {
    console.log("Insuficiente");
}
```

Estructura `switch`

El **switch** es otra manera de manejar múltiples condiciones. Es útil cuando tienes varias opciones para elegir.

Ejemplo de `switch`:

```
let dia = 2;

switch (dia) {
    case 1:
        console.log("Lunes");
        break;
    case 2:
        console.log("Martes");
        break;
    case 3:
        console.log("Miércoles");
        break;
    default:
        console.log("Día inválido");
}
```

3. Arrays en JavaScript

Los **arrays** son estructuras que permiten almacenar múltiples valores en una sola variable. Son útiles para manejar listas de datos, como números, cadenas de texto, objetos, y más. En JavaScript, los arrays son muy flexibles y ofrecen muchas funcionalidades.

Creación de Arrays

Puedes crear un array de varias maneras:

Usando la sintaxis de corchetes (`[ ]`):

```
let frutas = ["manzana", "banana", "naranja"];
```

Usando el constructor `Array()`:

```
let numeros = new Array(1, 2, 3, 4, 5);
```

Acceso a Elementos

Puedes acceder a los elementos de un array utilizando su índice. Recuerda que los índices en JavaScript comienzan en `0`.

```
let frutas = ["manzana", "banana", "naranja"];
console.log(frutas[0]); // Imprime "manzana"
```

Modificación de Elementos

Puedes modificar un elemento del array asignando un nuevo valor a un índice específico.

Ejemplo:

```
let frutas = ["manzana", "banana", "naranja"];
frutas[1] = "fresa"; // Cambia "banana" a "fresa"
console.log(frutas); // Imprime ["manzana", "fresa", "naranja"]
```

Métodos simples de Arrays

JavaScript ofrece varios métodos útiles para manipular arrays:

```
let frutas = ["manzana", "banana", "naranja"]; //Array de ejemplo
```

Método	Descripción	Ejemplo
push()	Agrega un elemento al final del array.	`frutas.push("kiwi");`
pop()	Elimina el último elemento del array.	`frutas.pop(); // Elimina "kiwi"`
shift():	Elimina el primer elemento del array.	`frutas.shift(); // Elimina "manzana"`
unshift()	Agrega un elemento al principio del array.	`frutas.unshift("mango");`
length	Propiedad que devuelve la cantidad de elementos en el array.	`console.log(frutas.length); // Imprime la cantidad de elementos`

indexOf()	Devuelve el primer índice en el que se puede encontrar un elemento en el array, o -1 si no se encuentra.	`let frutas = ['manzana', 'naranja', 'plátano'];`
		`let indice = frutas.indexOf('naranja');`
		`// 1`

includes()	Determina si un array incluye un determinado elemento, devolviendo true o false.	`let frutas = ['manzana', 'naranja', 'plátano'];` `let tieneNaranja = frutas.includes('naranja'); // true`

slice():	Devuelve una copia superficial de una porción del array en un nuevo array.	`let frutas = ['manzana', 'naranja', 'plátano'];`
		`let nuevasFrutas = frutas.slice(1, 3);`
		`// ['naranja', 'plátano']`

4. Ciclos en JavaScript

Los **ciclos** son estructuras que te permiten ejecutar un bloque de código varias veces. Son útiles cuando necesitas repetir acciones, como recorrer listas o realizar cálculos repetidos. En JavaScript, los ciclos más comunes son `for`, `while` y `do...while`.

Ciclo `for`

El ciclo `for` es ideal cuando sabes cuántas veces deseas ejecutar un bloque de código. Se compone de tres partes: inicialización, condición y expresión final.

Sintaxis:

```
for (inicialización; condición; expresión final) {
```

```
    // Código a ejecutar
}
```

Ejemplo:

```
for (let i = 0; i < 5; i++) {
    console.log("Número: " + i);
}
```

En este ejemplo, el ciclo se ejecuta 5 veces, desde `0` hasta `4`, imprimiendo el número en cada iteración.

Ciclo `while`

El ciclo `while` ejecuta un bloque de código mientras una condición sea verdadera. Es útil cuando no sabes cuántas veces se ejecutará el ciclo de antemano.

Sintaxis:

```
while (condición) {
    // Código a ejecutar
}
```

Ejemplo:

```
let contador = 0;

while (contador < 5) {
    console.log("Contador: " + contador);
    contador++; // Incrementa el contador
}
```

En este ejemplo, el ciclo continuará ejecutándose mientras `contador` sea menor que `5`.

Ciclo `do...while`

El ciclo `do...while` es similar al `while`, pero garantiza que el bloque de código se ejecute al menos una vez, ya que la condición se evalúa al final.

Sintaxis:

```
do {
    // Código a ejecutar
} while (condición);
```

Ejemplo:

```
let numero = 0;

do {
    console.log("Número: " + numero);
    numero++;
} while (numero < 5);
```

En este caso, el bloque se ejecuta primero y luego verifica si `numero` es menor que `5`.

5. Funciones en JavaScript

Las funciones son bloques de código diseñados para realizar una tarea específica. Te permiten encapsular lógica que puede ser reutilizada en diferentes partes de tu programa. Las funciones pueden aceptar parámetros y devolver resultados, lo que las convierte en herramientas poderosas para organizar y estructurar tu código.

Definición de una función

En JavaScript, puedes definir una función de varias maneras. La forma más común es usando la palabra clave `function`, seguida del nombre de la función, paréntesis y un bloque de código entre llaves.

Sintaxis:

```
function nombreFuncion(param1, param2) {
    // Código a ejecutar
    return param1+param2; // Opcional
}

console.log(nombreFuncion(5, 3))
```

Ejemplo de función

```
function sumar(a, b) {
    return a + b;
}

let resultado = sumar(5, 3);
console.log("El resultado es: " + resultado);
```

En este ejemplo, la función `sumar` toma dos **parámetros** (`a` y `b`), los suma y devuelve el resultado. Luego, llamamos a la función, le pasamos como **argumentos** los valores y mostramos el resultado en la consola.

Parámetros y **argumentos** son términos que a menudo se utilizan en el contexto de las funciones, pero tienen significados distintos.

Parámetros

- **Definición**: Los parámetros son variables que se definen en la declaración de una función. Actúan como marcadores de posición que recibirán valores cuando se llame a la función.
- **Ubicación**: Se especifican dentro de los paréntesis en la declaración de la función.
- **Uso**: Permiten a las funciones aceptar valores de entrada que pueden ser utilizados dentro de su bloque de código.

Ejemplo:

```
function sumar(a, b) {
    return a + b; // 'a' y 'b' son parámetros
}
```

En este caso, `a` y `b` son parámetros de la función `sumar`. Cuando defines la función, no tienen un valor específico; simplemente representan valores que la función podrá usar.

Argumentos

- **Definición**: Los argumentos son los valores reales que se pasan a una función cuando se llama. Son las instancias concretas de los parámetros.
- **Ubicación**: Se especifican en la llamada a la función, dentro de los paréntesis.
- **Uso**: Proporcionan a la función los datos que necesita para ejecutar su lógica.

```
let resultado = sumar(5, 3); // '5' y '3' son argumentos
```

En este caso, cuando llamas a la función `sumar`, estás pasando `5` y `3` como argumentos. Estos valores se asignan a los parámetros `a` y `b` en la función.

Retorno de valores

El uso de la instrucción `return` permite que una función devuelva un valor. Si no se especifica un `return`, la función devolverá `undefined`.

No todas las funciones **DEBEN** retornar un valor, aquellas funciones que no retornan ningún valor se definen cómo funciones tipo **void**.

Funciones anónimas

También puedes definir funciones sin un nombre, conocidas como funciones anónimas. Estas son comúnmente utilizadas como argumentos en otras funciones.

Ejemplo:

```
const multiplicar = function(x, y) {
    return x * y;
};

let producto = multiplicar(4, 2);
console.log("El producto es: " + producto);
```

Funciones de flecha

Las funciones de flecha son una forma más concisa de escribir funciones en JavaScript. Se introdujeron en ECMAScript 6 y son especialmente útiles para funciones pequeñas.

Sintaxis:

```
const nombreFuncion = (param1, param2) => {
    // Código a ejecutar
    return resultado; // Opcional
};
```

Ejemplo:

```
const restar = (x, y) => x - y;

let diferencia = restar(10, 4);
console.log("La diferencia es: " + diferencia);
```

Segunda parte - Programación orientada a objetos, matrices y lógica.

1. Clases en Javascript

ACLARACIÓN: Las clases en Javascript se introdujeron con Ecmascript 6 (el lenguaje de scripting en el que se basa Javascript). Pueden NO ser necesarias en muchos proyectos (orientados a la web), pero son FUNDAMENTALES para entender cómo funciona la programación orientada a objetos.

Una clase es una plantilla o modelo que define la estructura y el comportamiento de los objetos en programación orientada a objetos. En términos simples, una clase es como un plano que especifica qué propiedades y métodos tendrán los objetos que se crean a partir de ella.

En definitiva, una **CLASE** es el lienzo sobre el que pintamos **OBJETOS**.

Y los **OBJETOS** no son más que nuestros **PROPIOS** tipos de dato, que se componen de tipos de dato primitivos (string, number). Podemos crear objetos a conveniencia dependiendo de las necesidades que presente un problema, el ejemplo más común de objeto es Persona.

Una persona tiene muchas características, nombre, apellido, dirección, edad, DNI. Cada uno de esos datos individuales puede ser plasmado en una CLASE, y a su vez, con esa CLASE, podemos crear diferentes OBJETOS que representan fielmente a una persona en la vida real.

El **OBJETO** Persona equivaldría a: {'Juan', 'Perez', 'Mendoza', 20, 40000000}

Componentes de una clase

1. **Propiedades**: Son las características o atributos que describen el estado de un objeto. Por ejemplo, en una clase `Coche`, las propiedades pueden incluir `marca`, `modelo` y `color`.
2. **Métodos**: Son funciones definidas dentro de la clase que describen las acciones que un objeto puede realizar. Siguiendo el ejemplo anterior, un método podría ser `acelerar()` o `frenar()`.
3. **Constructor**: Es un método especial que se llama automáticamente al crear una nueva instancia de la clase. Se utiliza para inicializar las propiedades del objeto.

Ventajas de usar clases

- **Reutilización de código**: Las clases permiten crear múltiples instancias de un objeto con el mismo comportamiento sin duplicar código.
- **Encapsulamiento**: Las clases agrupan datos y métodos relacionados, lo que mejora la organización del código.
- **Herencia**: Las clases pueden heredar propiedades y métodos de otras clases, lo que facilita la creación de jerarquías y relaciones entre objetos.

Ejemplo:

```
class Coche {
    constructor(marca, modelo) {
        this.marca = marca;
        this.modelo = modelo;
    }

    mostrarInfo() {
        console.log(`Coche: ${this.marca} ${this.modelo}`);
    }
}

// Crear una instancia de la clase (es decir, un OBJETO}
let miCoche = new Coche("Toyota", "Corolla");

miCoche.mostrarInfo(); // Salida: Coche: Toyota Corolla
```

Ahora, vamos a ver un ejemplo de **RELACIÓN ENTRE DOS CLASES.**

Teniendo en cuenta la clase Coche que creamos anteriormente, vamos a crear una clase Persona. Esta clase Persona va a contener en su interior un Coche.

```
export class Persona {
  constructor(nombre, apellido) {
    this.nombre = nombre;
    this.apellido = apellido; //La Persona empieza sin coche
  }

  agregarCoche(coche) {
    //Con este método, usando this, le asignamos un Coche a la persona
    this.coche = coche;
  }

  mostrarInfo() {
    if (this.coche) {
      console.log(`Nombre completo: ${this.nombre} ${this.apellido}, tiene un coche
${this.coche.marca} ${this.coche.modelo}`);
    }
    else {
      console.log(`Nombre completo: ${this.nombre} ${this.apellido}`);
    }
  }
}
```

En el método principal:

```
//Creamos un OBJETO coche
let miCoche = new Coche("Toyota", "Corolla");

//Creamos un OBJETO persona
let persona = new Persona('Juan', 'Perez');

//Usamos el MÉTODO que creamos en PERSONA para AGREGAR un coche al OBJETO persona
persona.agregarCoche(miCoche)


persona.mostrarInfo(); //Se nos mostrarán los datos tanto de Persona cómo de Coche.
```

Y por último, vamos a ver el concepto de **HERENCIA** entre dos clases.

Herencia se refiere a una manera de compartir estructura entre clases, por ejemplo vamos a ver las clases Persona y Trabajador.

Persona tiene los atributos nombre y apellido.

Empleado, tiene los atributos nombre, apellido, trabajo y sueldo.

Para poder tener una estructura de datos más limpia, podemos hacer que Empleado herede los datos que comparte con Persona, en código se ve de la siguiente manera.

Por un lado tenemos la clase Persona.

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

  mostrarInfo() {
    console.log(`Nombre completo: ${this.nombre} ${this.apellido}`);
  }
}
```

Y por otro, la clase Empleado, que hereda con la palabra clave ‘extends’ los atributos y métodos de Persona.

```
export class Empleado extends Persona {
  constructor(nombre, apellido, puesto, salario) {
    super(nombre, apellido); // Llama al constructor de la clase Persona
    this.puesto = puesto;
    this.salario = salario;
  }

  mostrarInfo() {
    super.mostrarInfo(); // Llama al método de la clase padre
    console.log(`Puesto: ${this.puesto}, Salario: ${this.salario}`);
  }
}
```

Con el método super accedemos a los atributos y métodos de la clase padre, en este caso Persona.

Así usamos las clases:

```
const empleado1 = new Empleado('Ana', 'Gómez', 'Desarrolladora', 60000);

empleado1.mostrarInfo();
// Salida:
// Nombre completo: Ana Gómez
// Puesto: Desarrolladora, Salario: 60000
```

Una vez comprendidas las clases, vamos a dar paso a los objetos clásicos en Javascript.

2. Objetos clásicos en JavaScript

Los objetos, como vimos, son una de las estructuras de datos más importantes y versátiles en JavaScript. Permiten almacenar colecciones de datos y entidades complejas en una única variable. Un objeto es un conjunto de propiedades y métodos que pueden representar entidades del mundo real, como una persona, un coche o un producto.

Definición de un objeto

Un objeto se define como una colección de pares clave-valor, donde cada clave (o propiedad) es una cadena y el valor puede ser de cualquier tipo de dato, incluidos otros objetos y funciones.

```
let persona = {
    nombre: ‘Juan’,
    apellido: ‘Perez’,
    // ...
};
```

Métodos de un objeto

Los métodos son funciones que están asociadas a un objeto y se pueden invocar utilizando la notación de punto. Los métodos permiten que los objetos realicen acciones.

```
let coche = {
    marca: "Toyota",
    modelo: "Corolla",
    acelerar: function() {
        console.log("Acelerando...");
    }
};

coche.acelerar(); // Salida: Acelerando...
```

En este caso, `acelerar` es un método del objeto `coche`, y puedes llamarlo para ejecutar su lógica.

Objetos anidados (relación entre dos clases)

Los objetos pueden contener otros objetos, lo que permite crear estructuras de datos más complejas.

```
let estudiante = {
    nombre: "Ana",
    materias: {
        matematicas: "A",
        fisica: "B"
    }
```

```
};

console.log(estudiante.materias.matematicas); // Salida: A
```

Acceso a propiedades

Hay dos formas de acceder a las propiedades de un objeto: **notación de punto** y **notación de corchetes**.

```
// Notación de punto
console.log(estudiante.nombre);
console.log(estudiante.materias.fisica);

// Notación de corchetes
console.log(estudiante["nombre"]); // Notación de corchetes
console.log(estudiante[“materias”][“fisica”]);
```

Herencia en objetos clásicos.

```
// Objeto base
let Vehiculo = {
    marca: "Genérica",
    modelo: "Modelo",
    mostrarInfo: function() {
        console.log(`Marca: ${this.marca}, Modelo: ${this.modelo}`);
    }
};

// Crear un nuevo objeto que hereda de Vehiculo
let coche = Object.create(Vehiculo); // Herencia
coche.marca = "Toyota"; // Personalizamos la propiedad
coche.modelo = "Corolla";

// Agregamos un método específico para coche
coche.acelerar = function() {
    console.log("Acelerando...");
};

// Usar los métodos
coche.mostrarInfo(); // Salida: Marca: Toyota, Modelo: Corolla
coche.acelerar(); // Salida: Acelerando...
```

3. Arrays y bucles.

La iteración sobre arrays es una tarea común en programación, especialmente cuando necesitas procesar o manipular los elementos de un array. En JavaScript, puedes utilizar diferentes tipos de bucles para recorrer los elementos de un array y realizar operaciones sobre ellos.

A continuación vamos a revisar las formas más comunes de iterar sobre arrays en Javascript.

FOR:

```
let numeros = [1, 2, 3, 4, 5];

for (let i = 0; i < numeros.length; i++) {
    console.log("Número: " + numeros[i]);
}
```

Ahora vamos a recorrer cada valor, pero a la inversa, desde el 5 hasta el 1.

```
let numeros = [1, 2, 3, 4, 5];

for (let i = numeros.length - 1; i >= 0; i--) {
    console.log("Número: " + numeros[i]);
}
```

FOREACH:

```
const numeros = [1, 2, 3, 4, 5]

numeros.forEach((numero) => {
    console.log("Número: " + numero);
});
```

FOROF:

```
let numeros = [1, 2, 3, 4, 5];

for (let numero of numeros) {
    console.log("Número: " + numero);
}
```

A la inversa:

```
let numeros = [1, 2, 3, 4, 5];

for (let numero of numeros.reverse()) {
    console.log("Número: " + numero);
}
```

4. Funciones de transformación e iteración de arrays.

Javascript es un lenguaje de programación **FUNCIONAL**, por lo cuál cuenta con una gran variedad de funciones para trabajar con grandes cantidades de datos. Hay que tener en cuenta que un array puede tener cientos de miles de datos en su interior. A continuación tenemos una tabla que repasa los métodos funcionales más importantes.

Método	Descripción	Ejemplo
`map()`	Crea un nuevo array con los resultados de aplicar una función a cada elemento del array. También puede usarse sin función extra para iterar sobre el array.	`let numeros = [1, 2, 3];` `let dobles = numeros.map(num => num * 2); // [2, 4, 6]`
`filter()`	Crea un nuevo array con todos los elementos que pasen la prueba implementada por la función proporcionada.	`let numeros = [1, 2, 3, 4, 5];` `let pares = numeros.filter(num => num % 2 === 0); // [2, 4]`

`sort()`	Ordena los elementos de un array y los devuelve. Por defecto, los elementos se ordenan como cadenas.	`let numeros = [4, 2, 3, 1];` `numeros.sort(); // [1, 2, 3, 4]`
`reverse()`	Agrega un elemento al principio del array.	`let numeros = [1, 2, 3, 4];` `numeros.reverse(); // [4, 3, 2, 1]`
`forEach()`	Ejecuta una función proporcionada una vez por cada elemento del array.	`let frutas = ['manzana', 'naranja', 'plátano'];` `frutas.forEach(fruta => console.log(fruta)); // Imprime cada fruta`

5. Arrays de dos dimensiones (matrices).

¿Qué son las Matrices en JavaScript?

Las matrices (o arrays bidimensionales) son arrays que contienen otros arrays como elementos. Se utilizan para representar estructuras de datos en forma de tabla, donde cada elemento se puede acceder mediante dos índices: uno para la fila y otro para la columna.

Una matriz de 3x3 se puede representar gráficamente de esta manera:

0.0	0.1	0.2
1.0	1.1	1.2
2.0	2.1	2.2

Dónde cada número equivale a su respectivo índice (fila = Y, columna= X).

Ahora vamos a ver unos ejemplos de matrices directamente en código.

Definición simple:

```
let matriz = [
    [1, 2, 3],
    [4, 5, 6],
    [7, 8, 9]
];
```

Acceder a un elemento específico:

```
let valor = matriz[1][2];

// Accede al elemento en la fila 1, columna 2 (6 en este caso)
```

Recorrer matrices usando ciclos anidados:

Esto ya representa un ejercicio de lógica mayor, sin embargo es fundamental aprender a trabajar con matrices, ya que se usan cotidianamente, además de ser los ejercicios más comunes en entrevistas de trabajo.

```
for (let i = 0; i < matriz.length; i++) {

    let fila = ""; // Inicializa una cadena vacía para cada fila

    for (let j = 0; j < matriz[i].length; j++) {
        fila += matriz[i][j] + " "; // Agrega cada elemento seguido de un espacio
    }

    console.log(fila.trim()); // Imprime la fila, eliminando el espacio final
}


console.log(matriz.length) // Esto nos retorna cuanta cantidad de filas hay

//Se entiende que i equivale a un valor dentro de un ciclo
console.log(matriz[0].length) // Esto nos retorna cuanta cantidad de columnas hay
```