1. FUNCIONES EN JS

1.1. Formas de definir funciones

```
// Declaración (hoisting: se pueden usar antes de declararlas)
function sumar(a, b) {
  return a + b;
}

// Expresión
const restar = function(a, b) {
  return a - b;
};

// Función flecha
const multiplicar = (a, b) => a * b;
```

- **Declaración**: se “sube” (hoisting).
- **Expresión / flecha**: no se pueden usar antes de la línea donde se definen.

1.2. Funciones flecha (arrow functions)

```
const doble = n => n * 2;             // 1 parámetro, sin paréntesis
const saludar = () => "Hola";         // 0 parámetros, paréntesis
obligatorios
const suma = (a, b) => a + b;         // varios parámetros

// Con varias líneas, hay que usar llaves y return
const areaRectangulo = (ancho, alto) => {
  const area = ancho * alto;
  return area;
};
```

Importante:

- Arrow functions **no tienen su propio `this`**, ni `arguments`.
- Son ideales para:
    - callbacks (`map`, `filter`, `reduce`, eventos…)
    - funciones “cortas” y limpias

1.3. Parámetro rest y spread

```
// REST: agrupa en array
const sumarTodos = (...nums) => nums.reduce((acum, n) => acum + n,
0);

// SPREAD: “expande”
const arr1 = [1, 2, 3];
const arr2 = [...arr1, 4, 5]; // [1, 2, 3, 4, 5]
```

---

2. THIS, BIND Y ARROW FUNCTIONS

2.1. `this` en métodos de objetos

```
const persona = {
  nombre: "Ana",
  saludar() {
    console.log("Hola, soy", this.nombre);
  }
};

persona.saludar(); // "Hola, soy Ana" → this = persona
```

`this` apunta **al objeto que llama al método**.

2.2. Problema al separar la función

```
const persona = {
  nombre: "Ana",
  saludar() {
    console.log("Hola, soy", this.nombre);
  }
};

const otra = persona.saludar;
otra(); // this ya NO es persona → this.nombre = undefined
```

2.3. `bind` para fijar `this`

```
const saludarDeAna = persona.saludar.bind(persona);

saludarDeAna(); // "Hola, soy Ana" → this = persona aunque no se llame como persona.saludar()
```

`bind(obj)` → devuelve **una nueva función** donde `this` siempre será `obj`.

2.4. Arrow function y `this`

```
const persona = {
  nombre: "Lucía",
  decirNombre: () => {
    console.log(this.nombre);
  }
};

persona.decirNombre(); // this NO es persona → casi seguro undefined
```

- Arrow function **NO crea `this` propio**.
- Toma el `this` del lugar donde se define (scope léxico).
- Por eso **NO es buena idea** usar arrow para métodos que usan `this`.

2.5. Arrow + this en callbacks (caso útil)

```
const contador = {
  valor: 0,
  iniciar() {
    setInterval(() => {
      // Arrow: this es el mismo que en iniciar() → contador
      this.valor++;
      console.log(this.valor);
    }, 1000);
  }
};
```

En callbacks (`setTimeout`, `setInterval`, etc.) la flecha mantiene el `this` externo.

---

3. ARRAYS: MAP, FILTER, REDUCE

3.1. `map` – transforma elementos

```
const nums = [1, 2, 3];
const dobles = nums.map(n => n * 2); // [2, 4, 6]
```

- Devuelve **nuevo array**.
- Misma longitud.

3.2. `filter` – filtra elementos

```
const edades = [10, 20, 18, 30];
const mayores = edades.filter(e => e >= 18); // [20, 18, 30]
```

- Devuelve **nuevo array** con elementos que cumplan la condición.

3.3. `reduce` – reduce a un único valor

```
const nums = [1, 2, 3, 4];
const suma = nums.reduce((acum, n) => acum + n, 0); // 10
```

- `acum` → acumulador
- `n` → elemento actual
- `0` → valor inicial del acumulador

3.4. Ejemplo combinado con objetos

```
const alumnos = [
  { nombre: "Ana", nota: 8 },
  { nombre: "Luis", nota: 4 },
  { nombre: "María", nota: 6 }
];

const aprobados = alumnos.filter(a => a.nota >= 5);
const nombres = alumnos.map(a => a.nombre);
const media = alumnos.reduce((acum, a) => acum + a.nota, 0) /
alumnos.length;
```

---

4. OBJETOS Y POO

0. Crear arrays y cosas básicas

```
const numeros = [1, 2, 3, 4];
const frutas = ["manzana", "pera", "plátano"];

console.log(numeros.length);  // tamaño → 4
console.log(frutas[0]);       // "manzana"
console.log(frutas[frutas.length - 1]); // último elemento
```

---

1. Métodos que *modifican* el array

1.1. `push` y `pop` (final del array)

```
const arr = [1, 2, 3];

arr.push(4);        // añade al final → [1, 2, 3, 4]
const ultimo = arr.pop(); // quita y devuelve el último → 4, array: [1, 2, 3]
```

- `push(elem1, elem2, ...)` → añade al final
- `pop()` → quita el último y lo devuelve

1.2. `unshift` y `shift` (inicio del array)

```
const arr = [2, 3];

arr.unshift(1);          // añade al inicio → [1, 2, 3]
const primero = arr.shift(); // quita y devuelve el primero → 1, array: [2, 3]
```

1.3. `splice` (insertar / borrar en una posición)

Sintaxis básica:

```
array.splice(indiceInicio, cantidadAEliminar, elemOpcional1, elemOpcional2...)
```

Ejemplos:

```
const letras = ["a", "b", "c", "d"];

// Eliminar
letras.splice(1, 2); // desde índice 1, elimina 2 → elimina "b" y "c"
console.log(letras); // ["a", "d"]

// Insertar sin eliminar
const numeros = [1, 4, 5];
numeros.splice(1, 0, 2, 3); // en índice 1, borra 0 y añade 2, 3
```

```
console.log(numeros); // [1, 2, 3, 4, 5]
```

1.4. `sort` (ordenar)

□□ Por defecto ordena como **strings**.

```
const letras = ["b", "a", "c"];
letras.sort();           // ["a", "b", "c"]

const nums = [10, 2, 1];
nums.sort();             // ["1", "10", "2"] → [1, 10, 2] (mal)

nums.sort((a, b) => a - b); // correcto: [1, 2, 10]
```

1.5. `reverse` (invertir orden)

```
const arr = [1, 2, 3];
arr.reverse(); // [3, 2, 1]
```

---

2. Métodos que *no modifican* el array (devuelven otro)

2.1. `slice` (extraer una parte)

```
const letras = ["a", "b", "c", "d", "e"];

const parte = letras.slice(1, 4); // desde índice 1 hasta 4 (sin incluir)
console.log(parte);  // ["b", "c", "d"]
console.log(letras); // original sigue igual
```

- `slice(inicio, fin)` → copia desde `inicio` hasta `fin` (sin incluir).
- Si no pones `fin`, copia hasta el final.

2.2. `concat` (unir arrays)

```
const a = [1, 2];
const b = [3, 4];
const c = a.concat(b); // [1, 2, 3, 4]
```

2.3. `includes` (comprueba si existe un valor)

```
const frutas = ["manzana", "pera", "plátano"];

frutas.includes("pera");      // true
frutas.includes("naranja");   // false
```

2.4. `indexOf` y `lastIndexOf`

```
const letras = ["a", "b", "c", "b"];

letras.indexOf("b");      // 1 (primer "b")
letras.lastIndexOf("b");  // 3 (último "b")
letras.indexOf("z");      // -1 si no está
```

---

3. Métodos de “mapeo”: `forEach`, `map`, `filter`, `reduce`, `find`, `some`, `every`

3.1. `forEach` (recorre, pero no devuelve nada “útil”)

```
const nums = [1, 2, 3];
nums.forEach((n, i) => {
  console.log("Posición", i, "valor", n);
});
```

- Se usa para **recorrer** y hacer algo (pintar en pantalla, etc.)
- No devuelve un nuevo array con resultados.

3.2. `map` (transformar)

```
const nums = [1, 2, 3];

const dobles = nums.map(n => n * 2); // [2, 4, 6]

const usuarios = ["ana", "luis"];
const enMayus = usuarios.map(nombre => nombre.toUpperCase()); //
["ANA", "LUIS"]
```

3.3. `filter` (filtrar)

```
const nums = [1, 2, 3, 4, 5];

const pares = nums.filter(n => n % 2 === 0); // [2, 4]
const mayoresQueTres = nums.filter(n => n > 3); // [4, 5]
```

3.4. `reduce` (reducir a un valor)

```
const nums = [1, 2, 3, 4];

const suma = nums.reduce((acum, n) => acum + n, 0); // 10

const max = nums.reduce((maxActual, n) => n > maxActual ? n :
maxActual, nums[0]);
```

3.5. `find` (encuentra el primer elemento que cumple la condición)

```
const personas = [
  { nombre: "Ana", edad: 20 },
  { nombre: "Luis", edad: 17 },
  { nombre: "María", edad: 25 }
];

const mayorDeEdad = personas.find(p => p.edad >= 18);
console.log(mayorDeEdad); // { nombre: "Ana", edad: 20 } (el primero
que cumple)
```

3.6. `some` (al menos uno cumple)

```
const edades = [10, 15, 20];

edades.some(e => e >= 18); // true (hay al menos un mayor de edad)
edades.some(e => e < 5);   // false
```

3.7. `every` (todos cumplen)

```
const edades = [18, 20, 30];

edades.every(e => e >= 18); // true (todas ≥ 18)
edades.every(e => e > 18);  // false
```

---

4. Ejemplo típico de examen con arrays + objetos

```
const productos = [
  { nombre: "Teclado", precio: 20 },
  { nombre: "Ratón", precio: 10 },
  { nombre: "Monitor", precio: 150 },
];

// 1) Array solo con los nombres
const nombres = productos.map(p => p.nombre); // ["Teclado",
"Ratón", "Monitor"]

// 2) Filtrar productos de más de 20€
const caros = productos.filter(p => p.precio > 20); // [Monitor]

// 3) Precio total
const total = productos.reduce((acum, p) => acum + p.precio, 0); //
180

// 4) ¿Hay algún producto de menos de 15€?
const hayBarato = productos.some(p => p.precio < 15); // true

// 5) Buscar el primer producto que valga más de 50
const mayor50 = productos.find(p => p.precio > 50); // Monitor
```

4.1. Objetos literales

```
const persona = {
  nombre: "Ana",
  edad: 20,
  saludar() {
    console.log(`Hola, me llamo ${this.nombre}`);
  }
};
```

4.2. Funciones constructoras

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

Coche.prototype.descripcion = function() {
  return `${this.marca} ${this.modelo}`;
};

const c1 = new Coche("Toyota", "Corolla");
```

4.3. Clases ES6

```
class Rectangulo {
  constructor(ancho, alto) {
    this.ancho = ancho;
    this.alto = alto;
  }

  area() {
    return this.ancho * this.alto;
  }
}

class Cuadrado extends Rectangulo {
  constructor(lado) {
    super(lado, lado);
  }
}
```

4.4. Desestructuración básica

```
const usuario = { nombre: "Ana", edad: 20, ciudad: "Valencia" };
const { nombre, ciudad } = usuario;

const colores = ["rojo", "verde", "azul"];
const [primero, segundo] = colores;
```

---

5. DOM (Document Object Model)

DOM = representación del HTML en forma de objetos JS.

5.1. Seleccionar elementos

```
// Por id
const titulo = document.getElementById("titulo");

// Por clase
const items = document.getElementsByClassName("item");

// querySelector (1º que coincida)
const parrafo = document.querySelector("p.miClase");

// querySelectorAll (NodeList)
const liTodos = document.querySelectorAll("li");
```

5.2. Modificar contenido y estilos

```
const p = document.getElementById("texto");

p.textContent = "Nuevo texto";   // solo texto
p.innerHTML = "<b>Texto</b>";    // interpreta HTML

p.style.color = "red";           // CSS desde JS
p.classList.add("resaltado");    // añadir clase
p.classList.remove("oculto");
p.classList.toggle("activo");
```

5.3. Crear y añadir elementos

```
const ul = document.getElementById("lista");

const li = document.createElement("li");
li.textContent = "Elemento nuevo";
ul.appendChild(li); // lo añade al final
```

---

6. EVENTOS

6.1. Escuchar eventos

```
const btn = document.getElementById("btn");

// función normal
btn.addEventListener("click", function(event) {
  console.log("Has hecho click");
});

// arrow function
btn.addEventListener("click", (event) => {
  console.log("Click con arrow");
});
```

6.2. Eventos más comunes

- `click`
- `dblclick`
- `input` (cuando cambia el contenido de un input)
- `change` (cuando se confirma el cambio, ej. select)
- `submit` (formularios)
- `keydown`, `keyup`, `keypress` (teclado)

```
const input = document.getElementById("nombre");
input.addEventListener("input", () => {
  console.log(input.value);
});
```

6.3. Delegación de eventos

En vez de añadir un listener a cada hijo, lo pones en el **padre** y miras `event.target`.

```
const lista = document.getElementById("lista");

lista.addEventListener("click", (event) => {
  if (event.target.tagName === "LI") {
    console.log("Has pulsado:", event.target.textContent);
  }
});
```

Eventos de formulario

Se producen en los formularios:

- **focus** / **blur**: al obtener/perder el foco el elemento
- **change**: al perder el foco un <input> o <textarea> si ha cambiado su contenido o al cambiar de valor un <select> o un <checkbox>
- **input**: al cambiar el valor de un <imput> o <textarea> (se produce cada vez que escribimos una letra es estos elementos)
- **select**: al cambiar el valor de un <select> o al seleccionar texto de un <imput> o <textarea>
- **submit** / **reset**: al enviar/recargar un formulario

.type: qué evento se ha producido (click, submit, keyDown, ...)

- **.target**: es el elemento exacto donde se originó el evento — el que el usuario realmente tocó, hizo clic, etc.

- **.currentTarget**: es el elemento que tiene asociado el manejador del evento, es decir, quien está escuchando ese evento.

eventos de ratón:

    - **.button**: qué botón del ratón se ha pulsado (0: izq, 1: rueda; 2: dcho).
    - **.screenX** / **.screenY**: las corrdenadas del ratón respecto a la pantalla
    - **.clientX** / **.clientY**: las coordenadas del ratón respecto a la ventana cuando se produjo el evento
    - **.pageX** / **.pageY**: las coordenadas del ratón respecto al documento (si se ha hecho un scroll será el clientX/Y más el scroll)
    - **.offsetX** / **.offsetY**: las coordenadas del ratón respecto al elemento sobre el que se produce el evento
    - **.detail**: si se ha hecho click, doble click o triple click

Date:

`.getDay()`	Devuelve el día de la semana: OJO: `0` Domingo, `6` Sábado.
`.getFullYear()`	Devuelve el año con 4 cifras.
`.getMonth()`	Devuelve la representación interna del mes. OJO: `0` Enero - `11` Diciembre.
`.getDate()`	Devuelve el día del mes.
`.getHours()`	Devuelve la hora. OJO: Formato militar; `23` en lugar de `11`.
`.getMinutes()`	Devuelve los minutos.
`.getSeconds()`	Devuelve los segundos.
`.getMilliseconds()`	Devuelve los milisegundos.
`.getTime()`	Devuelve el UNIX Timestamp: segundos transcurridos desde `1/1/1970`.
`.getTimezoneOffset()`	Diferencia horaria (*en min*) de la hora local respecto a UTC (ver más adelante).

7. DATASET (`data-*`)

Sirve para guardar datos personalizados en el HTML y leerlos desde JS.

7.1. En HTML

```
<button id="btn" data-precio="20" data-nombre-
producto="Teclado">Comprar</button>
```

7.2. En JS

```
const btn = document.getElementById("btn");

btn.addEventListener("click", () => {
  const precio = Number(btn.dataset.precio);          // 20
  const nombre = btn.dataset.nombreProducto;          // "Teclado"
  console.log(nombre, precio);
});
```

Reglas:

- `data-precio` → `dataset.precio`
- `data-nombre-producto` → `dataset.nombreProducto` (guiones → camelCase)
- Todo se lee como **string**.

---

8. LOCALSTORAGE (persistencia en el navegador)

Guarda pares clave → valor **en el navegador**.
Siempre guarda strings.

8.1. Guardar y leer

```
// Guardar
localStorage.setItem("nombreUsuario", "Ana");
```

```
// Leer
const nombre = localStorage.getItem("nombreUsuario"); // "Ana"

// Borrar una clave
localStorage.removeItem("nombreUsuario");

// Borrar todo
localStorage.clear();
```

8.2. Guardar objetos / arrays (usar JSON)

```
const tareas = ["Estudiar", "Sacar al perro"];

localStorage.setItem("tareas", JSON.stringify(tareas));

// Luego...
const guardadas = localStorage.getItem("tareas");
const tareasRecuperadas = JSON.parse(guardadas); // vuelve a array
```

BOM

Window

`.name`: nombre de la ventana actual

- `.status`: valor de la barra de estado
- `.screenX`/`.screenY`: distancia de la ventana a la esquina izquierda/superior de la pantalla
- `.outerWidth`/`.outerHeight`: ancho/alto total de la ventana, incluyendo la toolbar y la scrollbar
- `.innerWidth`/`.innerHeight`: ancho/alto útil del documento, sin la toolbar y la scrollbar
- `.open(url, nombre, opciones)`: abre una nueva ventana. Devuelve el nuevo objeto ventana. Las principales opciones son:
  - `.toolbar`: si tendrá barra de herramientas
  - `.location`: si tendrá barra de dirección
  - `.directories`: si tendrá botones Adelante/Atrás

    - `.status`: si tendrá barra de estado
    - `.menubar`: si tendrá barra de menú
    - `.scrollbar`: si tendrá barras de desplazamiento
    - `.resizable`: si se puede cambiar su tamaño
    - `.width=px`/`.height=px`: ancho/alto
    - `.left=px`/`.top=px`: posición izq/sup de la ventana
- `.opener`: referencia a la ventana desde la que se abrió esta ventana (para ventanas abiertas con open)
- `.close()`: la cierra (pide confirmación, a menos que la hayamos abierto con open)
- `.moveTo(x,y)`: la mueve a las coord indicadas
- `.moveBy(x,y)`: la desplaza los px indicados
- `.resizeTo(x,y)`: la da el ancho y alto indicados
- `.resizeBy(x,y)`: le añade ese ancho/alto
- `.pageXoffset / pageYoffset`: scroll actual de la ventana horizontal / vertical

location

`.href`: devuelve la URL actual completa

- `.protocol`, `.hostname`, `.port`: devuelve el protocolo, host y puerto respectivamente de la URL actual
- `.pathname`, `hash`, `search`: devuelve la ruta al recurso actual, el gragmento (`#...`) y la cadena de búsqueda (`?...`) respectivamente
- `.reload()`: recarga la página actual
- `.assign(url)`: carga la página pasada como parámetro
- `.replace(url)`: ídem pero sin guardar la actual en el historial

history

`.length`: muestra el número de páginas almacenadas en el historial

- `.back()`: vuelve a la página anterior
- `.forward()`: va a la siguiente página
- `.go(num)`: se mueve num páginas hacia adelante o hacia atrás (si num es negativo) en el historial

navigator

`.userAgent`: muestra información sobre el navegador que usamos

- `.language`: muestra el idioma del navegador
- `.languages`: muestra los idiomas instalados en el navegador
- `.appVersion`: versión del navegador
- `.appName`: nombre del navegador
- `.appCodeName`: nombre en código del navegador
- `.product`: producto del navegador
- `.platform`: sistema en el que se ejecuta el navegador

9. MINI-RESUMEN EXPRESS (por si quieres repasar en 1 minuto)

- **Arrow functions**:
  `const f = (a, b) => a + b;`
  No tienen `this` propio.

- **this**:
  - En métodos normales (`saludar() {}`) → apunta al objeto que llama.
  - Arrow → usa el `this` exterior. No usar para métodos que usen `this`.

- **bind**:
  - `const f = persona.saludar.bind(persona);`
  - Fija `this = persona`.

- **map / filter / reduce**:
  - `map` → transforma array.

    - `filter` → se queda con algunos.
    - `reduce` → reduce a un valor.
- **DOM**:
    - `getElementById`, `querySelector`, `innerHTML`, `textContent`, `classList`.
- **Eventos**:
    - `addEventListener("click", ...)`
    - Delegación con `event.target`.
- **dataset**:
    - HTML: `data-precio="20"`
    - JS: `element.dataset.precio`
- **localStorage**:
    - `setItem`, `getItem`, `removeItem`, `clear`
    - Para objetos: `JSON.stringify` / `JSON.parse`.

1️⃣ `for` clásico

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

- **Inicialización:** `let i = 0`
- **Condición:** `i < 5`
- **Actualización:** `i++`
- Sirve para **iterar un número de veces conocido** o recorrer un array por índice.

Ejemplo con array

```
const numeros = [10, 20, 30];
for (let i = 0; i < numeros.length; i++) {
  console.log(numeros[i]);
}
```

2️⃣ `for...of` → recorrer valores de arrays o strings

```
const frutas = ["manzana", "pera", "plátano"];

for (const fruta of frutas) {
  console.log(fruta);
}

// Con strings
for (const letra of "Hola") {
  console.log(letra);
}
```

- Recorre **directamente los valores**, no los índices.
- Funciona con **arrays, strings y cualquier iterable**.

3️⃣ `for...in` → recorrer claves de objetos

```
const persona = {
  nombre: "Ana",
  edad: 25,
  ciudad: "Madrid"
};

for (const clave in persona) {
  console.log(clave, "→", persona[clave]);
}
```

- Recorre **las propiedades (claves)** de un objeto.

- No recomendado para arrays (puede recorrer propiedades heredadas y en desorden).
- Muy útil para objetos literales.

4️⃣ `forEach` → método de arrays

```
const numeros = [10, 20, 30];

numeros.forEach(function(n) {
  console.log(n);
});

// Con arrow function
numeros.forEach(n => console.log(n));

// Con índice y array completo
numeros.forEach((valor, indice, arr) => {
  console.log(`índice ${indice} → valor ${valor}`);
});
```

- Solo funciona con **arrays**.
- Ejecuta una **función para cada elemento**.
- No devuelve nada (`undefined`).

5️⃣ `while` → bucle con condición

```
let i = 0;
while (i < 5) {
  console.log(i);
  i++;
}
```

- Se ejecuta mientras la condición sea `true`.
- Útil cuando **no sabes cuántas veces se va a repetir** el bucle.
- La variable de control se inicializa fuera del bucle.

6️⃣ `do...while` → al menos una ejecución

```
let i = 0;
do {
  console.log(i);
  i++;
} while (i < 5);
```

- Se ejecuta **al menos una vez**, incluso si la condición inicial es `false`.
- Después de ejecutar, evalúa la condición para seguir iterando.

7️⃣ Comparación rápida

Bucle	Qué recorre	Cuándo usar
`for`	Índices de array o número	Sabes cuántas veces quieres iterar
`for...of`	Valores de arrays o iterables	Quieres solo los valores, no índices
`for...in`	Claves de objetos	Recorrer propiedades de objetos
`forEach`	Elementos de array	Ejecutar función por cada elemento
`while`	Según condición	No sabes cuántas veces, depende de condición
`do...while`	Según condición	Quieres ejecutar mínimo una vez