tema2: JavaScript Avanzado

Desarrollo Web en entorno cliente

José Antonio Torrado Navas

Noviembre 2024

Manejo avanzado de objetos.

Manejo avanzado de objetos.

- En javascript todos los elementos son objetos esto nos permite que cada objeto tenga una serie de atributos y funciones que nos permiten manejar el objeto.

Manejo avanzado de objetos.

- En javascript todos los elementos son objetos esto nos permite que cada objeto tenga una serie de atributos y funciones que nos permiten manejar el objeto.
- Vamos una serie de objetos que nos permiten hacer tareas de una forma mas sencilla.

Objeto `Math`

Math Redondeo de números a diferentes niveles

- `Math.round()`: Redondea al entero más cercano.
- `Math.ceil()`: Redondea hacia arriba.
- `Math.floor()`: Redondea hacia abajo.
- `Math.trunc()`: Elimina la parte decimal, devolviendo solo la parte entera.

Math: Redondeo de números a diferentes niveles

```
console.log(Math.round(7.4));
// Devuelve 7

console.log(Math.round(7.5));
// Devuelve 8

console.log(Math.ceil(7.2));
// Devuelve 8

console.log(Math.floor(7.8));
// Devuelve 7

console.log(Math.trunc(7.9));
// Devuelve 7 (solo la parte entera)
```

Math: Generación de números aleatorios

- Puedes usar `Math.random()` junto con `Math.floor()` o `Math.ceil()` para obtener números aleatorios en un rango determinado.

```
const randomNum1a10 = Math.floor(Math.random() * 10) + 1;
console.log(randomNum1a10);
// Número entero aleatorio entre 1 y 10

// Número entero aleatorio entre 5 y 15
const min = 5;
const max = 15;
const randomNum5a15 = Math.floor(Math.random() * (max - min + 1)) + min;
console.log(randomNum5a15);
```

Math: Calcular potencias y raíces

- `Math.pow(base, exponente)`: Eleva `base` al `exponente`.
- `Math.sqrt(x)`: Raíz cuadrada.
- `Math.cbrt(x)`: Raíz cúbica.

Math: Calcular potencias y raíces

```
console.log(Math.pow(3, 4));
// 3 elevado a 4: Devuelve 81
console.log(Math.sqrt(64));
// Raíz cuadrada de 64: Devuelve 8
console.log(Math.cbrt(27));
// Raíz cúbica de 27: Devuelve 3
```

Math Obtener el valor mínimo o máximo de una lista

- Usa `Math.min()` y `Math.max()` para encontrar el valor mínimo o máximo de una lista de números.

```
const numeros = [3, 7, 2, 8, -1, 4];

const minValor = Math.min(...numeros);
// Usa el operador spread para pasar los elementos
const maxValor = Math.max(...numeros);

console.log(minValor);
// Devuelve -1 (mínimo valor)
console.log(maxValor);
// Devuelve 8 (máximo valor)
```

objeto `Date`

Objeto `Date`

- El objeto **`Date`** en JavaScript es una herramienta para trabajar con fechas y horas.

Objeto `Date`

- El objeto **`Date`** en JavaScript es una herramienta para trabajar con fechas y horas.
- Con `Date`, puedes crear y manipular valores que representan momentos específicos en el tiempo, obtener la fecha y hora actuales.

Objeto `Date`

- El objeto **`Date`** en JavaScript es una herramienta para trabajar con fechas y horas.
- Con `Date`, puedes crear y manipular valores que representan momentos específicos en el tiempo, obtener la fecha y hora actuales.
- También puedes realizar operaciones como calcular la diferencia entre fechas o formatear una fecha en diferentes estilos.

Crear un objeto `Date`

Existen varias formas de crear un objeto `Date`:

1. **Fecha y hora actual**:

```
const ahora = new Date();
console.log(ahora);
// Devuelve la fecha y hora actuales
```

2. **Fecha específica**:

```
const fechaEspecifica = new Date('2024-12-25');
console.log(fechaEspecifica);
// Devuelve la fecha indicada
```

Crear un objeto `Date`

3. **Fecha con valores específicos (año, mes, día, hora, minutos, segundos, milisegundos)**:

```
const otraFecha = new Date(2024, 11, 25, 10, 30, 0);
// Meses: 0=enero, 11=diciembre
console.log(otraFecha);
// Devuelve la fecha y hora específica
```

4. **Fecha con milisegundos desde el 1 de enero de 1970**:

```
const fechaMilisegundos = new Date(0);
// Fecha en base a milisegundos
console.log(fechaMilisegundos);
// Devuelve: Thu Jan 01 1970 00:00:00 UTC
```

Métodos comunes del objeto `Date`

- El objeto `Date` proporciona varios métodos para obtener y manipular partes de una fecha:
  - **`getFullYear()`**: Obtiene el año completo.
  - **`getMonth()`**: Obtiene el mes (0 = enero, 11 = diciembre).
  - **`getDate()`**: Obtiene el día del mes.
  - **`getHours()`**: Obtiene la hora.
  - **`getMinutes()`**: Obtiene los minutos.
  - **`getSeconds()`**: Obtiene los segundos.
  - **`getMilliseconds()`**: Obtiene los milisegundos.
  - **`getDay()`**: Obtiene el día de la semana (0 = domingo, 6 = sábado).

Métodos comunes del objeto `Date`

```
const hoy = new Date();
console.log(hoy.getFullYear());
// Año actual
console.log(hoy.getMonth());
// Mes actual (0-11)
console.log(hoy.getDate());
// Día del mes actual
console.log(hoy.getDay());
// Día de la semana actual (0-6)
```

Establecer partes de una fecha:

- `setFullYear(año)`: Establece el año.
- `setMonth(mes)`: Establece el mes.
- `setDate(día)`: Establece el día del mes.
- `setHours(hora)`: Establece la hora.
- `setMinutes(minutos)`: Establece los minutos.
- `setSeconds(segundos)`: Establece los segundos.

Establecer partes de una fecha:

```
const fechaModificada = new Date();
fechaModificada.setFullYear(2025);
// Cambia el año a 2025
fechaModificada.setMonth(0);
// Cambia el mes a enero (0)
console.log(fechaModificada);
```

Comparar fechas:

- Puedes restar dos fechas para obtener la diferencia en milisegundos.
- También puedes comparar fechas directamente.

```
const fecha1 = new Date('2024-12-25');
const fecha2 = new Date('2024-12-31');
const diferencia = fecha2 - fecha1;
console.log(diferencia / (1000 * 60 * 60 * 24));
// Diferencia en días
```

Manejo avanzado de cadenas

Manejo avanzado de cadenas

- EL manejo de cadenas en cualquier lenguaje es fundamental. Normalmente trabajamos con texto que nos aporta el usuario, lo procesamos lo transformamos y mostramos texto nuevo.

Manejo avanzado de cadenas

- EL manejo de cadenas en cualquier lenguaje es fundamental. Normalmente trabajamos con texto que nos aporta el usuario, lo procesamos lo transformamos y mostramos texto nuevo.
- Es importante y javascript nos proporciona muchas funciones en el mismo objeto String para su manejo.

`split()` - Dividir una cadena en array

- La función **`split()`** divide una cadena en varias partes, creando un array a partir de esas partes.
- Toma un **separador** como argumento, que indica dónde se debe dividir la cadena.

```
const texto = "manzana,pera,plátano";
const frutas = texto.split(",");
// Divide la cadena por cada coma
console.log(frutas);
// Resultado: ["manzana", "pera", "plátano"]
```

- Si el separador es una cadena vacía (`""`), `split()` dividirá la cadena en cada carácter.

`concat()` - Concatenar cadenas

La función **`concat()`** une dos o más cadenas en una sola. Se puede usar para combinar varias cadenas.

```
const saludo = "Hola";
const nombre = "Juan";
const mensaje = saludo.concat(", ", nombre, "!");
console.log(mensaje);  // Resultado: "Hola, Juan!"
```

+ - Concatenar cadenas

- Otra forma de concatenar cadenas es con el operador +, que es más común:

```
const mensajeAlt = saludo + ", " + nombre + "!";
console.log(mensajeAlt);  // Resultado: "Hola, Juan!"
```

`replace()` - Reemplazar parte de una cadena

- La función **`replace()`** reemplaza una parte específica de la cadena con otro texto. Toma dos argumentos: el texto a reemplazar y el nuevo texto.

```
const frase = "La manzana es deliciosa";
const nuevaFrase = frase.replace("manzana", "pera");
console.log(nuevaFrase);
// Resultado: "La pera es deliciosa"
```

`replace()` - Reemplazar parte de una cadena

- `replace()` solo reemplaza la **primera** aparición del texto. Para reemplazar todas las apariciones, puedes usar una expresión regular con la bandera g (global).

```
const fraseMultiple = "La manzana es una manzana roja";
const fraseReemplazada = fraseMultiple.replace(/manzana/g,
console.log(fraseReemplazada);  // Resultado: "La pera es
```

##`toUpperCase()` y `toLowerCase()` - Cambiar mayúsculas y minúsculas

- **`toUpperCase()`** convierte toda la cadena a mayúsculas.
- **`toLowerCase()`** convierte toda la cadena a minúsculas.

```
const texto = "Hola Mundo";
console.log(texto.toUpperCase());
// Resultado: "HOLA MUNDO"
```

`trim()` - Eliminar espacios en blanco de los extremos

La función **`trim()`** elimina los espacios en blanco al inicio y al final de la cadena. Es útil para limpiar entradas de usuario.

```
const entrada = "   Hola Mundo   ";
console.log(entrada.trim());
// Resultado: "Hola Mundo"
```

- También existen `trimStart()` y `trimEnd()` para eliminar espacios solo al inicio o al final, respectivamente.

`substring()` y `slice()` - Extraer una parte de la cadena

- **`substring(inicio, fin)`**: Extrae una parte de la cadena desde la posición `inicio` hasta la posición `fin` (sin incluirla).
- **`slice(inicio, fin)`**: Similar a `substring()`, pero también acepta índices negativos, que cuentan desde el final de la cadena.

```
const texto = "JavaScript";
console.log(texto.substring(0, 4));
// Resultado: "Java"
console.log(texto.slice(0, 4));
// Resultado: "Java"
console.log(texto.slice(-6));
// Resultado: "Script" (cuenta desde el final)
```

substring() y slice() - Extraer una parte de la cadena

```
const url = "https://www.example.com";

// Extraer el nombre de dominio (sin "https://")
const dominio = url.substring(8, url.length);
console.log(dominio);
// "www.example.com"

// Extraer solo el nombre del sitio (sin "www.")
const sitio = dominio.slice(4);
console.log(sitio);
// "example.com"
```

`includes()` - Verificar si una cadena contiene otra cadena

La función **`includes()`** verifica si una cadena contiene una subcadena específica y devuelve `true` o `false`.

```
const texto = "Hoy es un buen día";
console.log(texto.includes("buen"));
// Resultado: true
console.log(texto.includes("noche"));
// Resultado: false
```

`startsWith()` y `endsWith()` - Verificar el inicio o final de una cadena

- **`startsWith()`**: Comprueba si la cadena empieza con un texto específico.
- **`endsWith()`**: Comprueba si la cadena termina con un texto específico.

```
const texto = "JavaScript es genial";
console.log(texto.startsWith("Java"));
// Resultado: true
console.log(texto.endsWith("genial"));
// Resultado: true
```

Resumen

- `split()`: Divide una cadena en un arreglo de subcadenas.
- `concat()`: Concadena dos o más cadenas.
- `replace()`: Reemplaza una parte específica de una cadena.
- `toUpperCase()` y `toLowerCase()`: Cambian el texto a mayúsculas o minúsculas.
- `trim()`: Elimina los espacios en blanco al inicio y al final.
- `charAt()` y `charCodeAt()`: Obtienen un carácter o su código ASCII.
- `substring()` y `slice()`: Extraen una parte de la cadena.
- `includes()`: Verifica si una cadena contiene otra subcadena.
- `startsWith()` y `endsWith()`: Verifican el inicio o final de una cadena.

Funciones como parámetros de otras funciones.

Funciones como parámetros de otras funciones.

- Un elemento importante de javascript es que todo son objetos. Esta característica permite hacer ciertas cosas de forma sencilla que otros lenguajes lo hacen de forma mas difícil.

Funciones como parámetros de otras funciones.

- Un elemento importante de javascript es que todo son objetos. Esta característica permite hacer ciertas cosas de forma sencilla que otros lenguajes lo hacen de forma mas difícil.
- En los lenguajes funcionales siempre se ha permitido usar funciones como parámetros de otras funciones.

Funciones como parámetros de otras funciones.

- Un elemento importante de javascript es que todo son objetos. Esta característica permite hacer ciertas cosas de forma sencilla que otros lenguajes lo hacen de forma mas difícil.
- En los lenguajes funcionales siempre se ha permitido usar funciones como parámetros de otras funciones.
- Algunos creadores de lenguajes les parecía confuso y lo quitaron por ejemplo java.

Funciones como parámetros de otras funciones.

- Un elemento importante de javascript es que todo son objetos. Esta característica permite hacer ciertas cosas de forma sencilla que otros lenguajes lo hacen de forma mas difícil.
- En los lenguajes funcionales siempre se ha permitido usar funciones como parámetros de otras funciones.
- Algunos creadores de lenguajes les parecía confuso y lo quitaron por ejemplo java.
- Google en el año 2000 creo el paradigma MapReduce para procesar grandes volúmenes de datos(Big data), se basa en tres funciones map, reduce y filter que tienen como parámetros funciones.

Funciones como parámetros de otras funciones.

- Un elemento importante de javascript es que todo son objetos. Esta característica permite hacer ciertas cosas de forma sencilla que otros lenguajes lo hacen de forma mas difícil.
- En los lenguajes funcionales siempre se ha permitido usar funciones como parámetros de otras funciones.
- Algunos creadores de lenguajes les parecía confuso y lo quitaron por ejemplo java.
- Google en el año 2000 creo el paradigma MapReduce para procesar grandes volúmenes de datos(Big data), se basa en tres funciones map, reduce y filter que tienen como parámetros funciones.
- En el bigdata han aparecido mas ejemplos de estos como sort y otros mas, ahora todos los lenguajes incorporan esto, javascript lo ha tenido desde el principio por lo que se ha adaptado muy bien

Ejemplo de función como parámetro de otra función

```
function procesarMensaje(mensaje, mostrar) {
  mostrar(mensaje);
}

// Función que muestra el mensaje en la consola
function mostrarEnConsola(texto) {
  console.log("Consola:", texto);
}

// Función que muestra el mensaje en una alerta
function mostrarEnAlerta(texto) {
  alert("Alerta: " + texto);
}

// Usamos `procesarMensaje` con diferentes funciones
procesarMensaje("Hola a todos", mostrarEnConsola);
// Muestra en consola
procesarMensaje("Bienvenidos", mostrarEnAlerta);
// Muestra en alerta
```

Mismo ejemplo con funciones flecha

```
function procesarMensaje(mensaje, mostrar) {
  mostrar(mensaje);
}

// Usamos `procesarMensaje` con diferentes funciones
procesarMensaje("Hola a todos", (texto)=>{
    console.log("Consola:", texto);
});
// Muestra en consola
procesarMensaje("Hola a todos", (texto)=>{
    alert("Consola:", texto);
});
// Muestra en alerta
```

Ejemplo II

```
function calcular(a, b, operacion) {
  return operacion(a, b);
}
// Función de suma
function suma(x, y) {
  return x + y;
}
// Función de resta
function resta(x, y) {
  return x - y;
}
// Usamos `calcular` con diferentes operaciones
console.log(calcular(5, 3, suma));
// 8
console.log(calcular(5, 3, resta));
// 2
```

Manejo avanzado de arrays

Manejo avanzado de arrays

- Los arrays es una estructura básica para guardar información y trabajar en un programa

Manejo avanzado de arrays

- Los arrays es una estructura básica para guardar información y trabajar en un programa
- Javascript incorpora objeto arrays una serie de métodos que nos permiten su manejo de forma mas sencilla.

Manejo avanzado de arrays

- Los arrays es una estructura básica para guardar información y trabajar en un programa
- Javascript incorpora objeto arrays una serie de métodos que nos permiten su manejo de forma mas sencilla.
- La mayoría de estos métodos usan alguna función como parámetro.

`forEach()`

- La función **`forEach()`** es un método que ejecuta una función proporcionada una vez para cada elemento del array.

`forEach()`

- La función **`forEach()`** es un método que ejecuta una función proporcionada una vez para cada elemento del array.
- A diferencia de otras funciones como `map()` o `filter()`, `forEach()` no devuelve un nuevo array, sino que simplemente realiza una operación en cada elemento del array.

`forEach()`

- La función **`forEach()`** es un método que ejecuta una función proporcionada una vez para cada elemento del array.
- A diferencia de otras funciones como `map()` o `filter()`, `forEach()` no devuelve un nuevo array, sino que simplemente realiza una operación en cada elemento del array.
- Es útil para realizar tareas en cada elemento, como imprimir datos, aplicar efectos, o modificar estructuras externas.

Sintaxis

```
array.forEach((elemento, índice, array) => {
  // código a ejecutar por cada elemento
});
```

- **`elemento`**: El elemento actual del array.
- **`índice`** (opcional): La posición del elemento actual en el array.
- **`array`** (opcional): El array que está siendo recorrido.

Ejemplo I

Sumar los valores de un array y almacenarlo en una variable externa

```
const numeros = [5, 10, 15];
let suma = 0;

numeros.forEach((numero) => {
  suma += numero;
  // Se acumula la suma en la variable externa
});

console.log("La suma de los números es:", suma);
// 30
```

Modificar objetos en un array

- Imaginemos que tenemos una lista de productos y queremos aplicar un descuento del 10% a cada producto. `forEach()` nos permite modificar cada elemento directamente.

```
const productos = [
  { nombre: "Laptop", precio: 1000 },
  { nombre: "Teclado", precio: 50 },
  { nombre: "Monitor", precio: 200 }
];

productos.forEach((producto) => {
  producto.precio *= 0.9;
  // Aplica un descuento del 10%
});

console.log(productos);
/* Resultado:
[
  { nombre: "Laptop", precio: 900 },
  { nombre: "Teclado", precio: 45 },
  { nombre: "Monitor", precio: 180 }
]
*/
```

Ejemplo

- **Contar la cantidad de elementos que cumplen una condición**
- En este ejemplo, usamos `forEach()` para contar cuántos números en un array son mayores a 10.

```
const numeros = [5, 12, 8, 20, 15];
let contador = 0;

numeros.forEach((numero) => {
  if (numero > 10) {
    contador++;
  }
});

console.log("Números mayores a 10:", contador);  // 3
```

`map()`

- La función `map()` en JavaScript crea un **nuevo array** al aplicar una función de transformación a cada elemento del array original.

`map()`

- La función **`map()`** en JavaScript crea un **nuevo array** al aplicar una función de transformación a cada elemento del array original.
- `map()` no modifica el array original, sino que devuelve uno nuevo en el que cada elemento es el resultado de la función aplicada.

`map()`

- La función `map()` en JavaScript crea un **nuevo array** al aplicar una función de transformación a cada elemento del array original.
- `map()` no modifica el array original, sino que devuelve uno nuevo en el que cada elemento es el resultado de la función aplicada.
- Es útil cuando quieres transformar **todos** los elementos de un array, como convertir a mayúsculas, aplicar cálculos o convertir objetos.

Sintaxis

```
const nuevoArray = array.map((elemento, índice, array) => {
  // código para transformar cada elemento
});
```

- **`elemento`**: El elemento actual en el array.
- **`índice`** (opcional): La posición del elemento en el array.
- **`array`** (opcional): El array que está siendo recorrido.

Ejemplo I

- **Multiplicar cada número por un factor específico**
- Supongamos que tenemos un array de precios y queremos crear uno nuevo aplicando un impuesto del 20% a cada precio.

```
const precios = [100, 200, 300];
const preciosConImpuesto = precios.map((precio) => precio *

console.log(preciosConImpuesto);  // [120, 240, 360]
```

Ejemplo II

- **Convertir una lista de nombres a mayúsculas** `En este ejemplo,map()‘` se usa para crear un nuevo array en el que cada nombre está en mayúsculas.

```
const nombres = ["Ana", "Carlos", "Beatriz"];
const nombresMayusculas = nombres.map((nombre) => nombre.to

console.log(nombresMayusculas);
// ["ANA", "CARLOS", "BEATRIZ"]
```

Ejemplo III

- **Transformar un array de objetos en un nuevo formato**
- Supongamos que tenemos una lista de productos y queremos transformar cada producto en un nuevo objeto que incluya el nombre en mayúsculas y el precio con descuento.

```
const productos = [
  { nombre: "Laptop", precio: 1000 },
  { nombre: "Teclado", precio: 50 },
  { nombre: "Monitor", precio: 300 }
];

const productosDescuento = productos.map((producto) => ({
  nombre: producto.nombre.toUpperCase(),
  precioDescuento: producto.precio * 0.9
}));

console.log(productosDescuento);
/* Resultado:
[
  { nombre: "LAPTOP", precioDescuento: 900 },
  { nombre: "TECLADO", precioDescuento: 45 },
  { nombre: "MONITOR", precioDescuento: 270 }
]
*/
```

Ejemplo IV

- **Combinar dos arrays en un nuevo formato**
- Si tienes dos arrays, puedes combinarlos en uno solo usando `map()`. En este caso, combinamos un array de nombres y uno de edades en un array de objetos.

```
const nombres = ["Ana", "Carlos", "Beatriz"];
const edades = [25, 30, 35];

const personas = nombres.map((nombre, i) => ({
  nombre: nombre,
  edad: edades[i]
}));

console.log(personas);
/* Resultado:
[
  { nombre: "Ana", edad: 25 },
  { nombre: "Carlos", edad: 30 },
  { nombre: "Beatriz", edad: 35 }
]
*/
```

`filter()`

- La función `filter()` en JavaScript crea un **nuevo array** con todos los elementos que cumplen una condición específica, determinada por una función de retorno.

`filter()`

- La función `filter()` en JavaScript crea un **nuevo array** con todos los elementos que cumplen una condición específica, determinada por una función de retorno.
- `filter()` no modifica el array original y solo incluye en el nuevo array aquellos elementos que pasen la condición.

`filter()`

- La función `filter()` en JavaScript crea un **nuevo array** con todos los elementos que cumplen una condición específica, determinada por una función de retorno.
- `filter()` no modifica el array original y solo incluye en el nuevo array aquellos elementos que pasen la condición.
- Es ideal para crear subarrays basados en características particulares de los elementos.

Sintaxis

```
const nuevoArray = array.filter((elemento, índice, array) =
  // código que retorna true o false
});
```

- **elemento**: El elemento actual del array.
- **índice** (opcional): La posición del elemento en el array.
- **array** (opcional): El array que está siendo recorrido.

Ejemplo I

- **Filtrar números mayores a un valor específico** ‘En este ejemplo, creamos un nuevo array que contiene solo los números mayores a 10.

```
const numeros = [5, 12, 8, 130, 44];
const mayoresA10 = numeros.filter((numero) => numero > 10);

console.log(mayoresA10);
// [12, 130, 44]
```

Ejemplo II

- **Filtrar objetos en un array según una propiedad**
- Aquí tenemos una lista de usuarios y queremos filtrar solo los que son mayores de edad.

```
const usuarios = [
  { nombre: "Juan", edad: 17 },
  { nombre: "María", edad: 22 },
  { nombre: "Pedro", edad: 20 }
];

const mayoresDeEdad = usuarios.filter((usuario) => usuario.edad >= 18);
console.log(mayoresDeEdad);
/* Resultado:
[
  { nombre: "María", edad: 22 },
  { nombre: "Pedro", edad: 20 }
]
*/
```

Ejemplo III

- **Filtrar un array de fechas para obtener solo las del futuro**
- En este ejemplo, creamos un array con solo las fechas que son posteriores a la fecha actual.

```
const fechas = [
  new Date("2024-10-10"),
  new Date("2025-01-01"),
  new Date("2023-12-31")
];

const fechaActual = new Date();
const fechasFuturas = fechas.filter((fecha) => fecha > fechaActu

console.log(fechasFuturas);
// Fechas futuras según la fecha actual
```

`reduce()`

- La función **`reduce()`** en JavaScript aplica una función a un acumulador y a cada elemento del array (de izquierda a derecha) para reducir el array a un solo valor.

`reduce()`

- La función **`reduce()`** en JavaScript aplica una función a un acumulador y a cada elemento del array (de izquierda a derecha) para reducir el array a un solo valor.
- A diferencia de otros métodos como `map()` o `filter()`, que devuelven arrays, `reduce()` devuelve un solo valor, que puede ser un número, una cadena, un objeto o cualquier otra estructura.

Sintaxis

```
const resultado = array.reduce((acumulador, elemento, índice, array) => {
  // código que realiza la operación de reducción
}, valorInicial);
```

- `acumulador`: El valor acumulado que se actualiza en cada iteración.
- `elemento`: El elemento actual del array.
- `índice` (opcional): La posición del elemento en el array.
- `array` (opcional): El array que está siendo recorrido.
- `valorInicial`: El valor inicial del acumulador (opcional). Si se omite, el primer elemento del array será el valor inicial.

Ejemplos I

- **Sumar todos los elementos de un array**
- El uso más común de `reduce()` es sumar todos los elementos de un array.

```
const numeros = [5, 10, 15];
const sumaTotal = numeros.reduce((acumulador, numero) => acumulador + numero, 0);

console.log(sumaTotal);  // 30
```

Ejemplo II

- **Contar la cantidad de ocurrencias de cada elemento en un array**
- En este ejemplo, `reduce()` se usa para crear un objeto que cuenta cuántas veces aparece cada elemento en un array.

```
const frutas = ["manzana", "naranja", "manzana", "pera", "naranja", "manzana"];
const conteoFrutas = frutas.reduce((contador, fruta) => {
  contador[fruta] = (contador[fruta] || 0) + 1;
  return contador;
}, {});

console.log(conteoFrutas);
// Resultado: { manzana: 3, naranja: 2, pera: 1 }
```

`find()`

- La función `find()` en JavaScript devuelve el **primer elemento** de un array que cumple con la condición especificada en la función de retorno.

`find()`

- La función **`find()`** en JavaScript devuelve el **primer elemento** de un array que cumple con la condición especificada en la función de retorno.
- Si no encuentra ningún elemento que cumpla con la condición, devuelve `undefined`.

`find()`

- La función **`find()`** en JavaScript devuelve el **primer elemento** de un array que cumple con la condición especificada en la función de retorno.
- Si no encuentra ningún elemento que cumpla con la condición, devuelve `undefined`.
- `find()` es útil cuando quieres encontrar un único elemento que cumpla un criterio específico, como buscar un objeto en un array según su propiedad.

Sintaxis

```
const resultado = array.find((elemento, índice, array) => {
  // código que retorna true o false
});
```

- **`elemento`**: El elemento actual del array.
- **`índice`** (opcional): La posición del elemento en el array.
- **`array`** (opcional): El array que está siendo recorrido.

Ejemplo I

- **Encontrar el primer número mayor que un valor específico**
- En este ejemplo, queremos encontrar el primer número que sea mayor que 10 en un array de números.

```
const numeros = [4, 9, 11, 7, 15];
const mayorA10 = numeros.find((numero) => numero > 10);

console.log(mayorA10);  // 11
```

Ejemplo II

- **Buscar un objeto en un array por su propiedad**
  Aquí,find()se usa para buscar el primer objeto que tenga unid‘ específico.

```
const usuarios = [
  { id: 1, nombre: "Juan" },
  { id: 2, nombre: "Ana" },
  { id: 3, nombre: "Luis" }
];

const usuario = usuarios.find((usuario) => usuario.id === 2
console.log(usuario);  // { id: 2, nombre: "Ana" }
```

Ejemplo II

- **Buscar el primer número en un rango específico**
- En este ejemplo, `find()` se usa para encontrar el primer número que esté entre 10 y 20 en un array.

```
const numeros = [3, 8, 12, 17, 25];
const numeroEnRango = numeros.find((numero) => numero >= 10 && numero <= 20);

console.log(numeroEnRango);  // 12
```

`sort()`

- La función **`sort()`** en JavaScript se utiliza para ordenar los elementos de un array.

`sort()`

- La función **`sort()`** en JavaScript se utiliza para ordenar los elementos de un array.
- De forma predeterminada, `sort()` convierte los elementos en cadenas y los ordena en orden lexicográfico (alfabético), lo cual puede producir resultados inesperados al ordenar números.

`sort()`

- La función **`sort()`** en JavaScript se utiliza para ordenar los elementos de un array.
- De forma predeterminada, `sort()` convierte los elementos en cadenas y los ordena en orden lexicográfico (alfabético), lo cual puede producir resultados inesperados al ordenar números.
- Para un control más específico sobre el orden, puedes pasar una función de comparación personalizada que defina cómo se deben ordenar los elementos.

Sintaxis

```
array.sort((a, b) => {
  // función de comparación que devuelve:
  // un número negativo si "a" debe estar antes que "b"
  // cero si "a" y "b" son iguales
  // un número positivo si "a" debe estar después de "b"
});
```

- La función sort espera que la función de comparación devuelva específicamente un número (negativo, cero o positivo) para determinar el orden entre los elementos a y b

Ejemplo I

- **Ordenar números en orden ascendente y descendente**
- Por defecto, `sort()` no ordena los números de forma numérica correcta (debido a la conversión a cadenas), por lo que necesitamos una función de comparación.

```
const numeros = [3, 1, 4, 1, 5, 9];

// Orden ascendente
numeros.sort((a, b) => a - b);
console.log("Ascendente:", numeros);
// [1, 1, 3, 4, 5, 9]

// Orden descendente
numeros.sort((a, b) => b - a);
console.log("Descendente:", numeros);
// [9, 5, 4, 3, 1, 1]
```

Ejemplo II

- **Ordenar cadenas alfabéticamente e ignorando mayúsculas/minúsculas**
- En este ejemplo, ordenamos un array de palabras ignorando las mayúsculas y minúsculas.

```
const palabras = ["Banana", "manzana", "Cereza", "fresa"];
palabras.sort((a, b) => a.toLowerCase().localeCompare(b.toLowerCase()));

console.log("Orden alfabético:", palabras);
// ["Banana", "Cereza", "fresa", "manzana"]
```

- El método .localeCompare() es una función de JavaScript que compara dos cadenas en un orden específico de idioma (llamado “locale”)

Ejemplo III

- **Ordenar objetos en función de una propiedad numérica**
- Si tienes un array de objetos, puedes ordenarlos por una propiedad específica. Aquí ordenamos productos por precio de menor a mayor.

```
const productos = [
  { nombre: "Laptop", precio: 1000 },
  { nombre: "Teclado", precio: 50 },
  { nombre: "Monitor", precio: 200 }
];

productos.sort((a, b) => a.precio - b.precio);
console.log("Orden por precio:", productos);
/* Resultado:
[
  { nombre: "Teclado", precio: 50 },
  { nombre: "Monitor", precio: 200 },
  { nombre: "Laptop", precio: 1000 }
]
*/
```

Ejemplo IV

- **Ordenar objetos por propiedad de texto**
- Podemos ordenar objetos por una propiedad de texto, como el nombre, en orden alfabético.

```
const productos = [
  { nombre: "Laptop", precio: 1000 },
  { nombre: "Teclado", precio: 50 },
  { nombre: "Monitor", precio: 200 }
];

productos.sort((a, b) => a.nombre.localeCompare(b.nombre));
console.log("Orden alfabético por nombre:", productos);
/* Resultado:
[
  { nombre: "Laptop", precio: 1000 },
  { nombre: "Monitor", precio: 200 },
  { nombre: "Teclado", precio: 50 }
]
*/
```