Array en JavaScript

1. Introducción a los Arrays

Los arrays son una de las estructuras de datos que almacenan una coleccón de elementos ordenados como números, cadenas, objetos o incluso otros arrays. Los arrays en JavaScript son mutables y su tamaño puede cambiar dinámicamente.

Características principales

1. **Indexación por Cero:** Los elementos de un array se numeran desde el índice 0.

   > Ejemplo: En `[10, 20, 30]`, el primer elemento es `10` y está en el índice `0`.

2. **Tipos de Datos Mixtos:** Los arrays pueden almacenar valores de diferentes tipos (números, strings, booleanos, objetos, etc.).

   > Ejemplo: `[42, "Hola", true, { clave: "valor" }]`.

3. **Propiedad length:** Proporciona la cantidad de elementos en el array.

   > Ejemplo: En `[1, 2, 3]`, haciendo `array.length = 3`.

4. **Dinamismo:** Pueden cambiar de tamaño dinámicamente, añadiendo o eliminando elementos.

5. **Métodos Incorporados:** Los arrays tienen métodos útiles para manipulación, como `push`, `pop`, `map`, y más.

Declaración y creación de arrays

1. **Forma Literal `[]`**

   Es la forma más común y recomendada para crear arrays.

   ```
   // Crear un array vacío
   let arrayVacio = [];

   // Crear un array con elementos
   let frutas = ["manzana", "naranja", "plátano"];
   console.log(frutas[0]); // "manzana"
   console.log(frutas.length); // 3
   ```

2. **Con el Constructor `new Array()`**

   Una forma alternativa, aunque menos utilizada.

   ```
   // Crear un array vacío
   let arrayVacio = new Array();
   ```

```
// Crear un array con elementos
let numeros = new Array(1, 2, 3, 4);
console.log(numeros[2]); // 3

// Crear un array con un tamaño específico
let arrayConTamaño = new Array(5); // Crea un array con 5 espacios
vacíos
console.log(arrayConTamaño.length); // 5
```

Nota: Usar `new Array(5)` puede ser confuso, ya que no crea un array con valores `undefined`, sino un array con "espacios vacíos".

Diferencias entre Arrays y Objetos

Característica	Array	Objeto
Estructura	Colección ordenada basada en índices numéricos.	Colección desordenada basada en claves.
Acceso	Acceso por índice: `array[0]`.	Acceso por clave: `obj["clave"]` o `obj.clave`.
Propósito Principal	Listas de elementos.	Representar entidades con propiedades y valores.
Métodos Incorporados	Métodos como `push`, `pop`, `map`, etc., para manipular.	Métodos básicos como `Object.keys`.
Iteración	Ideal para iteraciones con bucles y métodos como `forEach`.	Menos optimizado para listas grandes.

Ejemplo de un array frente a un objeto:

```
// Array
let frutas = ["manzana", "naranja", "plátano"];
console.log(frutas[1]); // "naranja"

// Objeto
let fruta = { nombre: "manzana", color: "rojo" };
console.log(fruta.color); // "rojo"
```

2. Propiedades Básicas de los Arrays

Propiedad `length`

La propiedad `length` devuelve la cantidad de elementos en un array. Es una propiedad dinámica: se actualiza automáticamente cuando se agregan o eliminan elementos del array.

Ejemplos:

```
let numeros = [10, 20, 30, 40];
console.log(numeros.length); // 4

// Modificar el tamaño del array
numeros.length = 2; // Reduce el array
console.log(numeros); // [10, 20]

numeros.length = 5; // Expande el array (los nuevos índices serán
"vacíos")
console.log(numeros); // [10, 20, <3 vacíos>]
```

Índices en los Arrays

Cada elemento en un array tiene un índice basado en su posición.

- Los índices comienzan en `0` (indexación por cero).
- Puedes acceder, modificar o eliminar elementos usando estos índices.

Ejemplo:

```
let colores = ["rojo", "verde", "azul"];
console.log(colores[0]); // "rojo"
console.log(colores[2]); // "azul"

// Modificar un elemento
colores[1] = "amarillo";
console.log(colores); // ["rojo", "amarillo", "azul"]

// Agregar un elemento en un índice específico
colores[3] = "violeta";
console.log(colores); // ["rojo", "amarillo", "azul", "violeta"]

// Acceder a un índice fuera de rango
console.log(colores[10]); // undefined
```

Tipos de Datos que Pueden Almacenar

Los arrays en JavaScript pueden contener cualquier tipo de datos, incluso mezclados en el mismo array. Esto incluye:

- **Primitivos**: Números, cadenas, booleanos, `null`, `undefined`, `BigInt`, `Symbol`.
- **Referenciados**: Objetos, arrays, funciones, etc.

Ejemplo:

```
let datosMixtos = [
  42,
  "Hola",
```

```
  true,
  null,
  undefined,
  { clave: "valor" },
  [1, 2, 3],
];
console.log(datosMixtos[5].clave); // "valor"
console.log(datosMixtos[6][1]); // 2
```

Arrays Multidimensionales

Un array multidimensional es un array que contiene otros arrays como elementos. Los más comunes son los **arrays bidimensionales**, que se usan para representar matrices o tablas.

Creación y Acceso:

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

// Acceder a elementos
console.log(matriz[0][1]); // 2 (fila 0, columna 1)
console.log(matriz[2][2]); // 9 (fila 2, columna 2)

// Modificar un elemento
matriz[1][1] = 50;
console.log(matriz[1]); // [4, 50, 6]
```

Operaciones con Arrays Multidimensionales:

```
// Recorrer todos los elementos
for (let i = 0; i < matriz.length; i++) {
  for (let j = 0; j < matriz[i].length; j++) {
    console.log(matriz[i][j]);
  }
}

// Agregar una nueva fila
matriz.push([10, 11, 12]);
console.log(matriz);
```

Ejemplo de Uso Práctico de Arrays Multidimensionales:

Representar un tablero de juego (como el de Tic Tac Toe):

```
let tablero = [
  ["X", "O", "X"],
  ["O", "X", "O"],
  ["X", "O", "X"],
];

console.log(tablero[1][2]); // "O" (fila 1, columna 2)
```

3. Métodos Básicos para Manipular Arrays

Agregar y Eliminar Elementos

a) `push()` y `pop()`

- `push()`: Agrega uno o más elementos al **final** del array.
- `pop()`: Elimina el último elemento del array y lo devuelve.

Ejemplo:

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

// Agregar elementos al final
numeros.push(4, 5);
console.log(numeros); // [1, 2, 3, 4, 5]

// Eliminar el último elemento
let eliminado = numeros.pop();
console.log(eliminado); // 5
console.log(numeros); // [1, 2, 3, 4]
```

b) `unshift()` y `shift()`

- `unshift()`: Agrega uno o más elementos al **inicio** del array.
- `shift()`: Elimina el primer elemento del array y lo devuelve.

Ejemplo:

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

// Agregar elementos al inicio
frutas.unshift("plátano", "uva");
console.log(frutas); // ["plátano", "uva", "manzana", "naranja"]

// Eliminar el primer elemento
let eliminado = frutas.shift();
console.log(eliminado); // "plátano"
console.log(frutas); // ["uva", "manzana", "naranja"]
```

Acceso y Modificación de Elementos

Uso de índices

- Puedes acceder a elementos usando su índice y modificarlos directamente.

Ejemplo:

```
let colores = ["rojo", "verde", "azul"];

// Acceder
console.log(colores[1]); // "verde"

// Modificar
colores[1] = "amarillo";
console.log(colores); // ["rojo", "amarillo", "azul"]
```

Método `splice()`

- `splice()`: Permite agregar, eliminar o reemplazar elementos en una posición específica.

Sintaxis:

```
array.splice(índice, cantidad, elementosOpcionales);
```

Ejemplos:

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

// Eliminar elementos
numeros.splice(1, 2); // Elimina 2 elementos desde el índice 1
console.log(numeros); // [1, 4, 5]

// Reemplazar elementos
numeros.splice(1, 1, 10, 20); // Reemplaza 1 elemento en el índice 1 con
10 y 20
console.log(numeros); // [1, 10, 20, 5]

// Insertar sin eliminar
numeros.splice(2, 0, 30);
console.log(numeros); // [1, 10, 30, 20, 5]
```

Crear Copias y Subarrays

a) `slice()`

- `slice()`: Crea una copia de una porción del array sin modificar el original.
- **Sintaxis:** `array.slice(inicio, fin)`
  - `inicio`: Índice donde comienza la copia.
  - `fin`: Índice donde termina la copia (no incluido).

Ejemplo:

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

// Copiar desde el índice 1 hasta antes del índice 4
let subArray = numeros.slice(1, 4);
console.log(subArray); // [2, 3, 4]

// Copiar desde el índice 2 hasta el final
let copia = numeros.slice(2);
console.log(copia); // [3, 4, 5]

// Copiar todo el array
let copiaCompleta = numeros.slice();
console.log(copiaCompleta); // [1, 2, 3, 4, 5]
```

b) Operador Spread (...)

- **Spread**: Permite crear copias y combinar arrays de forma más concisa.

Ejemplo:

```
let numeros = [1, 2, 3];
let copia = [...numeros]; // Crear una copia
console.log(copia); // [1, 2, 3]

// Combinar arrays
let nuevosNumeros = [...numeros, 4, 5, 6];
console.log(nuevosNumeros); // [1, 2, 3, 4, 5, 6]

// Duplicar y modificar elementos
let duplicados = [...numeros, ...numeros];
console.log(duplicados); // [1, 2, 3, 1, 2, 3]
```

4. Iterar sobre Arrays

Iterar sobre un array significa recorrer cada elemento del mismo para realizar una operación o analizar sus valores. Aquí tienes formas tradicionales y modernas de hacerlo.

Uso de Bucles Tradicionales

a) `for`

El bucle `for` es una forma básica y flexible de iterar sobre los elementos de un array.

Sintaxis:

```
for (let i = 0; i < array.length; i++) {
  // Operaciones con array[i]
}
```

Ejemplo:

```
let numeros = [10, 20, 30, 40];

for (let i = 0; i < numeros.length; i++) {
  console.log(`Índice ${i}: ${numeros[i]}`);
}
// Salida:
// Índice 0: 10
// Índice 1: 20
// Índice 2: 30
// Índice 3: 40
```

b) `while`

El bucle `while` itera mientras se cumpla una condición.

Ejemplo:

```
let numeros = [10, 20, 30, 40];
let i = 0;

while (i < numeros.length) {
  console.log(`Índice ${i}: ${numeros[i]}`);
  i++;
}
// Salida:
// Índice 0: 10
// Índice 1: 20
// Índice 2: 30
// Índice 3: 40
```

Métodos Modernos para Iteración

a) `forEach()`

- `forEach()` es un método que recorre cada elemento de un array y ejecuta una función proporcionada.
- Es ideal para ejecutar una acción en cada elemento, pero **no retorna un nuevo array**.

Sintaxis:

```
array.forEach((elemento, índice, array) => {
  // Operaciones con elemento
});
```

Ejemplo:

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

frutas.forEach((fruta, index) => {
  console.log(`Fruta en índice ${index}: ${fruta}`);
});
// Salida:
// Fruta en índice 0: manzana
// Fruta en índice 1: naranja
// Fruta en índice 2: plátano
```

Nota: No puedes usar `break` en `forEach()` para detener la iteración. Si necesitas esa funcionalidad, utiliza `for`.

b) `map()`

- `map()` es similar a `forEach()`, pero **retorna un nuevo array** con los resultados de aplicar una función a cada elemento.
- Es útil cuando quieres transformar los elementos de un array.

Sintaxis:

```
let nuevoArray = array.map((elemento, índice, array) => {
  // Operaciones con elemento
  return nuevoElemento;
});
```

Ejemplo:

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

// Crear un nuevo array multiplicando cada número por 2
let numerosDuplicados = numeros.map((numero) => numero * 2);
console.log(numerosDuplicados); // [2, 4, 6, 8]
```

Otro Ejemplo:

```
let nombres = ["Ana", "Luis", "María"];

// Transformar cada nombre en un objeto
let nombresObj = nombres.map((nombre) => {
  return { nombre: nombre, longitud: nombre.length };
});
console.log(nombresObj);
// Salida:
// [
//   { nombre: "Ana", longitud: 3 },
//   { nombre: "Luis", longitud: 4 },
//   { nombre: "María", longitud: 5 }
// ]
```

Comparación entre Métodos Tradicionales y Modernos

Método	Usos Principales	Características
`for`	Iteraciones personalizadas.	Más control, pero menos conciso.
`while`	Iteraciones con condiciones dinámicas.	Flexible, pero puede ser menos claro.
`forEach()`	Realizar acciones en cada elemento.	Más legible, no retorna nuevo array.
`map()`	Transformar elementos en un nuevo array.	Retorna un nuevo array transformado.

5. Métodos Avanzados de Arrays

Búsqueda y Filtro

`find()`

- **Busca el primer elemento** en un array que cumpla con una condición específica.
- Retorna el elemento encontrado o `undefined` si no lo encuentra.

```
let numeros = [5, 12, 8, 130, 44];

let mayor10 = numeros.find((num) => num > 10);
console.log(mayor10); // 12
```

`filter()`

- Retorna un **nuevo array** con todos los elementos que cumplan una condición.

```
let numeros = [5, 12, 8, 130, 44];

let mayores10 = numeros.filter((num) => num > 10);
console.log(mayores10); // [12, 130, 44]
```

`includes()`

- Verifica si un array contiene un elemento específico.
- Retorna `true` o `false`.

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

console.log(frutas.includes("naranja")); // true
console.log(frutas.includes("uva")); // false
```

`some()` y `every()`

- `some()`: Verifica si **al menos un elemento** cumple con una condición.
  Retorna `true` si encuentra un elemento que cumple la condición.
- `every()`: Verifica si **todos los elementos** cumplen con una condición.
  Retorna `true` solo si todos los elementos cumplen.

```
let numeros = [5, 12, 8, 130, 44];

// ¿Algún número mayor que 100?
console.log(numeros.some((num) => num > 100)); // true

// ¿Todos los números son mayores que 3?
console.log(numeros.every((num) => num > 3)); // true
```

Orden y Transformación

`sort()`

- Ordena los elementos de un array en su lugar.
- Por defecto, ordena como cadenas (alfabéticamente).

Ejemplo de Ordenación Alfabética:

```
let frutas = ["manzana", "naranja", "plátano"];
frutas.sort();
console.log(frutas); // ["manzana", "naranja", "plátano"]
```

Ejemplo de Ordenación Numérica:

```
let numeros = [40, 1, 5, 200];

// Orden ascendente
numeros.sort((a, b) => a - b);
console.log(numeros); // [1, 5, 40, 200]

// Orden descendente
numeros.sort((a, b) => b - a);
console.log(numeros); // [200, 40, 5, 1]
```

`reverse()`

- Invierte el orden de los elementos de un array.

Ejemplo:

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

`reduce()` y `reduceRight()`

- **`reduce()`**: Aplica una función a un acumulador y cada elemento del array, de izquierda a derecha, para reducirlo a un único valor.
- **`reduceRight()`**: Igual que `reduce()`, pero de derecha a izquierda.

Ejemplo con `reduce()`:

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

// Sumar todos los elementos
let suma = numeros.reduce((acumulador, actual) => acumulador + actual, 0);
console.log(suma); // 10
```

Ejemplo con `reduceRight()`:

```
let palabras = ["uno", "dos", "tres"];
let frase = palabras.reduceRight(
  (acumulador, actual) => acumulador + " " + actual
);
console.log(frase); // "tres dos uno"
```

Conversión de Arrays

`join()`

- Convierte los elementos de un array en un string, separados por un delimitador especificado.

Ejemplo:

```
let palabras = ["Hola", "mundo", "JavaScript"];
let frase = palabras.join(" ");
console.log(frase); // "Hola mundo JavaScript"
```

`toString()`

- Convierte el array en un string, separando los elementos con comas.

Ejemplo:

```
let numeros = [1, 2, 3, 4];
console.log(numeros.toString()); // "1,2,3,4"
```

Resumen de Métodos

Método	Propósito
`find()`	Encuentra el primer elemento que cumple una condición.
`filter()`	Crea un nuevo array con elementos que cumplen una condición.
`includes()`	Verifica si un elemento está presente en el array.
`some()`	Comprueba si al menos un elemento cumple una condición.
`every()`	Comprueba si todos los elementos cumplen una condición.
`sort()`	Ordena los elementos del array.
`reverse()`	Invierte el orden de los elementos del array.
`reduce()`	Reduce los elementos del array a un único valor.
`reduceRight()`	Igual que `reduce()`, pero de derecha a izquierda.
`join()`	Combina los elementos en un string con un delimitador.
`toString()`	Convierte el array en un string separado por comas.

6. Arrays Multidimensionales

Un array multidimensional es un array que contiene otros arrays como elementos. Cada array interno puede considerarse como una "fila" o "columna". La estructura más común es el **array bidimensional**, pero

puedes anidar arrays a cualquier nivel.

Imagina un array bidimensional como una tabla con filas y columnas:

```
let matriz = [
  [1, 2, 3], // Fila 0
  [4, 5, 6], // Fila 1
  [7, 8, 9], // Fila 2
];
```

En este caso:

- El índice del primer nivel (0, 1, 2) corresponde a las filas.
- El índice del segundo nivel (0, 1, 2) corresponde a las columnas.

Para acceder al número `6`, usas `matriz[1][2]`:

- `1` = segunda fila.
- `2` = tercera columna.

Creación de Arrays Anidados

Puedes crear arrays multidimensionales de diferentes maneras:

Definición Literal

Define el array directamente con arrays internos:

```
let matriz = [
  [1, 2, 3],
  [4, 5, 6],
  [7, 8, 9],
];
console.log(matriz[0][1]); // 2
```

Usando Bucles

Puedes construir un array bidimensional dinámicamente:

```
let filas = 3;
let columnas = 3;
let matriz = [];

// Crear la matriz
for (let i = 0; i < filas; i++) {
  let fila = [];
  for (let j = 0; j < columnas; j++) {
    fila.push(i * columnas + j + 1); // Generar números secuenciales
```

```
  }
  matriz.push(fila);
}

console.log(matriz);
// [
//   [1, 2, 3],
//   [4, 5, 6],
//   [7, 8, 9]
// ]
```

Acceso y Modificación de Elementos

En arrays multidimensionales, cada nivel de anidación requiere un índice adicional:

```
let matriz = [
  [10, 20, 30],
  [40, 50, 60],
  [70, 80, 90],
];

// Acceder al número 50 (fila 1, columna 1)
console.log(matriz[1][1]); // 50

// Modificar el número 80 (fila 2, columna 1)
matriz[2][1] = 100;
console.log(matriz);
// [
//   [10, 20, 30],
//   [40, 50, 60],
//   [70, 100, 90]
// ]
```

Métodos Avanzados para Arrays Multidimensionales

Aplanar un Array con `flat()`

Si tienes un array anidado y necesitas convertirlo en un array plano (sin niveles de anidación), usa `flat()`.

```
let arrayAnidado = [1, [2, 3], [[4, 5], 6]];

// Aplanar un nivel
console.log(arrayAnidado.flat(1)); // [1, 2, 3, [4, 5], 6]

// Aplanar todos los niveles
console.log(arrayAnidado.flat(Infinity)); // [1, 2, 3, 4, 5, 6]
```

Mapear y Aplanar con `flatMap()`

Combina la transformación de elementos y el aplanamiento.

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

// Mapear y duplicar cada número
let resultado = numeros.flatMap((num) => [num, num * 2]);
console.log(resultado); // [1, 2, 2, 4, 3, 6]
```

Ejercicios Prácticos

Sumar los Elementos de una Matriz

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

// Usar flat() y reduce()
let suma = matriz.flat().reduce((acumulador, actual) => acumulador +
actual, 0);
console.log(suma); // 45
```

Crear una Matriz de Multiplicación

```
let tamaño = 5;
let matriz = [];

for (let i = 1; i <= tamaño; i++) {
  let fila = [];
  for (let j = 1; j <= tamaño; j++) {
    fila.push(i * j); // Multiplicación
  }
  matriz.push(fila);
}

console.log(matriz);
// [
//   [1, 2, 3, 4, 5],
//   [2, 4, 6, 8, 10],
//   [3, 6, 9, 12, 15],
//   [4, 8, 12, 16, 20],
//   [5, 10, 15, 20, 25]
// ]
```

Encontrar el Máximo Valor de una Matriz

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

// Aplanar y encontrar el máximo
let maximo = Math.max(...matriz.flat());
console.log(maximo); // 9
```

7. Arrays y Destructuración

La destructuración de arrays es una forma elegante y concisa de asignar valores de un array a variables individuales. Es ampliamente utilizada en JavaScript moderno para simplificar el código y hacerlo más legible.

Introducción a la Destructuración

Concepto Básico

La destructuración de arrays permite extraer valores y asignarlos a variables con una sintaxis más breve.

Sintaxis:

```
const [variable1, variable2, ...] = array;
```

Ejemplo:

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

// Asignar valores del array a variables
const [primero, segundo, tercero] = colores;
console.log(primero); // "rojo"
console.log(segundo); // "verde"
console.log(tercero); // "azul"
```

Omisión de Elementos

Puedes omitir valores usando comas.

Ejemplo:

```

```

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

// Omitir el segundo elemento
const [primero, , tercero] = numeros;
console.log(primero); // 1
console.log(tercero); // 3
```

Valores Predeterminados

Si un array no tiene suficientes valores, puedes asignar valores predeterminados.

Ejemplo:

```
const frutas = ["manzana"];

// Usar valores predeterminados
const [primera, segunda = "naranja"] = frutas;
console.log(primera); // "manzana"
console.log(segunda); // "naranja"
```

Destructuración en Parámetros de Funciones

Concepto Básico

Puedes utilizar destructuración directamente en los parámetros de las funciones para trabajar con arrays.

Ejemplo:

```
function imprimirColores([color1, color2]) {
  console.log(`Color 1: ${color1}`);
  console.log(`Color 2: ${color2}`);
}

const colores = ["rojo", "verde", "azul"];
imprimirColores(colores);
// Salida:
// Color 1: rojo
// Color 2: verde
```

Con Valores Predeterminados

Si el array no tiene suficientes elementos, puedes combinar destructuración con valores predeterminados.

Ejemplo:

```

```

```
function sumar([a = 0, b = 0]) {
  return a + b;
}

console.log(sumar([5, 10])); // 15
console.log(sumar([5])); // 5
console.log(sumar([])); // 0
```

Uso Avanzado con Arrays Anidados

Puedes trabajar con arrays dentro de arrays utilizando destructuración anidada.

Ejemplo:

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

const [[a, b], [c, d]] = numeros;
console.log(a, b, c, d); // 1 2 3 4
```

Uso Combinado con el Operador Rest (`...`)

El operador **rest** (`...`) permite capturar el resto de los elementos de un array en una variable.

Capturar Elementos Restantes

Puedes asignar los primeros valores a variables específicas y capturar el resto en un array.

Ejemplo:

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

const [primero, segundo, ...resto] = numeros;
console.log(primero); // 1
console.log(segundo); // 2
console.log(resto); // [3, 4, 5]
```

Combinar con Funciones

El operador rest es útil para trabajar con funciones que aceptan un número variable de argumentos.

Ejemplo:

```
function sumarTodos([primero, segundo, ...otros]) {
  const sumaOtros = otros.reduce((acc, num) => acc + num, 0);
  return primero + segundo + sumaOtros;
}

console.log(sumarTodos([10, 20, 30, 40])); // 100
console.log(sumarTodos([5, 10])); // 15
```

Uso Avanzado: Reestructuración

El operador rest también puede ser utilizado para dividir y reorganizar un array.

Ejemplo:

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

// Separar los primeros dos números y reorganizar
const [primero, segundo, ...otros] = numeros;
const nuevoArray = [...otros, primero, segundo];
console.log(nuevoArray); // [3, 4, 5, 1, 2]
```

Prácticas Avanzadas con Destructuración

Intercambiar Valores

La destructuración puede facilitar el intercambio de valores entre variables.

Ejemplo:

```
let a = 10,
  b = 20;

// Intercambiar valores
[a, b] = [b, a];
console.log(a); // 20
console.log(b); // 10
```

Destructuración Dinámica con Datos JSON

La destructuración también es útil para trabajar con arrays generados dinámicamente, como datos provenientes de APIs.

Ejemplo:

```
const datos = {
  usuario: "Juan",
  favoritos: ["JavaScript", "Python", "Go"],
};

const {
  favoritos: [primero, segundo],
} = datos;
console.log(primero); // "JavaScript"
console.log(segundo); // "Python"
```

Resumen

Característica	Uso
Asignación básica	Extraer valores de un array y asignarlos a variables.
Omisión de elementos	Saltar valores específicos del array.
Valores predeterminados	Asignar valores por defecto si no existen en el array.
Destructuración en funciones	Recibir arrays como parámetros y descomponerlos directamente.
Uso del operador rest (`...`)	Capturar el resto de los elementos del array.
Reestructuración y reorganización	Crear nuevos arrays reorganizando los valores existentes.

8. Arrays y Programación Funcional

Los arrays en JavaScript son fundamentales para trabajar en un estilo de programación funcional. Este enfoque se basa en funciones puras y evita cambios en el estado global, lo que ayuda a crear código más predecible, modular y reutilizable. Aquí veremos cómo usar arrays en este paradigma, desde un nivel básico hasta avanzado.

Creación de Pipelines con Métodos Encadenados

Los pipelines consisten en encadenar métodos para aplicar transformaciones o procesos secuenciales sobre los datos de un array. Este enfoque es claro y conciso.

Nivel Junior: Encadenamiento Básico

Usar métodos como `map()`, `filter()`, y `reduce()` para realizar transformaciones paso a paso.

Ejemplo:

```
// Dado un array de números, filtrar los pares, multiplicarlos por 2 y
sumar el total
let numeros = [1, 2, 3, 4, 5, 6];
```

```
let resultado = numeros
  .filter((num) => num % 2 === 0) // Filtra pares: [2, 4, 6]
  .map((num) => num * 2) // Multiplica por 2: [4, 8, 12]
  .reduce((suma, num) => suma + num, 0); // Suma total: 24

console.log(resultado); // 24
```

Nivel Intermedio: Optimización y Legibilidad

Crear pipelines más legibles al usar funciones auxiliares nombradas.

Ejemplo:

```
let numeros = [1, 2, 3, 4, 5, 6];

// Funciones auxiliares
const esPar = (num) => num % 2 === 0;
const multiplicarPorDos = (num) => num * 2;

// Pipeline
let resultado = numeros
  .filter(esPar) // [2, 4, 6]
  .map(multiplicarPorDos) // [4, 8, 12]
  .reduce((suma, num) => suma + num, 0); // 24

console.log(resultado); // 24
```

Nivel Avanzado: Reutilización y Composición

Crear funciones reutilizables y genéricas que permiten construir pipelines flexibles.

Ejemplo:

```
let numeros = [1, 2, 3, 4, 5, 6];

// Función para componer funciones
const compose =
  (...funcs) =>
  (value) =>
    funcs.reduce((acc, func) => func(acc), value);

// Funciones auxiliares
const filtrarPares = (arr) => arr.filter((num) => num % 2 === 0);
const duplicar = (arr) => arr.map((num) => num * 2);
const sumar = (arr) => arr.reduce((suma, num) => suma + num, 0);

// Pipeline reutilizable
const procesarNumeros = compose(filtrarPares, duplicar, sumar);
```

```
console.log(procesarNumeros(numeros)); // 24
```

Transformaciones Complejas Usando `map()` y `reduce()`

Nivel Junior: Operaciones Simples con `map()`

Ejemplo: Convertir un array de objetos en un array con solo un atributo.

```
let usuarios = [
  { nombre: "Ana", edad: 25 },
  { nombre: "Luis", edad: 30 },
  { nombre: "María", edad: 28 },
];

let nombres = usuarios.map((usuario) => usuario.nombre);
console.log(nombres); // ["Ana", "Luis", "María"]
```

Nivel Intermedio: Sumarización con `reduce()`

Ejemplo: Obtener el total de edades de los usuarios.

```
let usuarios = [
  { nombre: "Ana", edad: 25 },
  { nombre: "Luis", edad: 30 },
  { nombre: "María", edad: 28 },
];

let totalEdad = usuarios.reduce((suma, usuario) => suma + usuario.edad,
0);
console.log(totalEdad); // 83
```

Nivel Avanzado: Agrupamiento con `reduce()`

Ejemplo: Agrupar usuarios por su grupo de edad.

```
let usuarios = [
  { nombre: "Ana", edad: 25 },
  { nombre: "Luis", edad: 30 },
  { nombre: "María", edad: 28 },
  { nombre: "Pedro", edad: 34 },
  { nombre: "Carla", edad: 25 },
];

let grupos = usuarios.reduce((acc, usuario) => {
  let grupo = usuario.edad < 30 ? "Jóvenes" : "Adultos";
```

```
  if (!acc[grupo]) acc[grupo] = [];
  acc[grupo].push(usuario.nombre);
  return acc;
}, {});

console.log(grupos);
// {
//   Jóvenes: ["Ana", "María", "Carla"],
//   Adultos: ["Luis", "Pedro"]
// }
```

Aplicaciones Prácticas

Procesar Datos de una API

Simular datos de una API y transformarlos para obtener información útil.

Ejemplo:

```
let productos = [
  { id: 1, nombre: "Laptop", precio: 800, disponible: true },
  { id: 2, nombre: "Teléfono", precio: 500, disponible: false },
  { id: 3, nombre: "Tablet", precio: 300, disponible: true },
];

// Obtener nombres de productos disponibles
let disponibles = productos
  .filter((producto) => producto.disponible)
  .map((producto) => producto.nombre);

console.log(disponibles); // ["Laptop", "Tablet"]
```

Calcular Estadísticas

Ejemplo: Calcular el promedio de edades.

```
let edades = [25, 30, 28, 34, 25];

let promedio = edades.reduce((suma, edad) => suma + edad, 0) /
edades.length;
console.log(promedio); // 28.4
```

Procesar Texto

Ejemplo: Contar palabras únicas en un texto.

```
let texto = "JavaScript es genial y JavaScript es poderoso";

let palabrasUnicas = texto
  .toLowerCase()
  .split(" ")
  .reduce((acc, palabra) => {
    if (!acc.includes(palabra)) acc.push(palabra);
    return acc;
  }, []);

console.log(palabrasUnicas); // ["javascript", "es", "genial", "y",
"poderoso"]
```

9. Casos Especiales y Buenas Prácticas con Arrays

1. Diferencias entre Valores Primitivos y Referencias

Valores Primitivos

- Los valores primitivos en JavaScript incluyen `number`, `string`, `boolean`, `null`, `undefined`, `symbol`, y `bigint`.
- **Se pasan por valor**, lo que significa que al asignarlos o pasarlos a funciones, se copia el valor.

Ejemplo:

```
let a = 10;
let b = a; // Se copia el valor
b = 20;

console.log(a); // 10
console.log(b); // 20
```

Referencias

- Los arrays y objetos son tipos **referenciales**, lo que significa que se pasan por referencia.
- Cambiar la referencia afecta al array original.

Ejemplo:

```
let array1 = [1, 2, 3];
let array2 = array1; // Ambas variables apuntan al mismo array

array2[0] = 10;
console.log(array1); // [10, 2, 3]
console.log(array2); // [10, 2, 3]
```

2. Clonación: Profunda vs. Superficial

Clonación Superficial

- Copia solo el nivel superior del array. Si contiene objetos o arrays anidados, la referencia se mantiene.

Métodos para clonación superficial:

- **Operador Spread:**

```
let original = [1, 2, 3];
let copia = [...original];

copia[0] = 10;
console.log(original); // [1, 2, 3]
console.log(copia); // [10, 2, 3]
```

- **Método `slice()`:**

```
let original = [1, 2, 3];
let copia = original.slice();

copia[1] = 20;
console.log(original); // [1, 2, 3]
console.log(copia); // [1, 20, 3]
```

Limitación: Si el array tiene objetos anidados, estos se copiarán por referencia:

```
let original = [{ id: 1 }, { id: 2 }];
let copia = [...original];

copia[0].id = 99;
console.log(original[0].id); // 99
```

Clonación Profunda

- Copia todos los niveles de un array, creando nuevas referencias en lugar de mantener las originales.

Métodos para clonación profunda:

- `JSON.parse(JSON.stringify(array))`:

```
let original = [{ id: 1 }, { id: 2 }];
let copia = JSON.parse(JSON.stringify(original));
```

```
copia[0].id = 99;
console.log(original[0].id); // 1
```

Nota: Este método no soporta funciones o valores especiales como `undefined`.

- **Librerías externas** (como Lodash con `_.cloneDeep()`):
  Ideal para clonaciones profundas en proyectos avanzados.

3. Evitar Mutaciones No Deseadas

Es importante evitar cambios accidentales en el array original.
Buenas prácticas:

- **Usar métodos inmutables:** Prefiere métodos que no modifiquen el array original (`map()`, `filter()`, `slice()`) en lugar de los que lo hacen (`push()`, `splice()`, `sort()`).

Ejemplo:

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

// Método mutable
numeros.push(4);
console.log(numeros); // [1, 2, 3, 4]

// Método inmutable
let nuevoArray = [...numeros, 4];
console.log(numeros); // [1, 2, 3]
console.log(nuevoArray); // [1, 2, 3, 4]
```

4. Manejo de Arrays Vacíos y Valores `undefined`

Arrays Vacíos

Trabajar con un array vacío puede ser problemático si no se manejan adecuadamente los casos especiales.

Ejemplo de problemas:

```
let arrayVacio = [];

// Buscar el primer elemento
console.log(arrayVacio[0]); // undefined

// Uso de métodos que dependen de elementos
console.log(arrayVacio.map((x) => x * 2)); // []
```

Buenas prácticas:

- Verifica si el array tiene elementos antes de procesarlo:

```
if (arrayVacio.length > 0) {
  console.log("El array no está vacío.");
} else {
  console.log("El array está vacío.");
}
```

Valores `undefined`

Un array puede tener elementos no definidos (`undefined`), ya sea porque no se asignaron o se dejaron explícitamente.

Ejemplo:

```
let array = [1, undefined, 3];
console.log(array[1]); // undefined
```

Cómo evitarlos:

- Asegúrate de inicializar todos los elementos del array.
- Usa métodos que filtren o manejen valores `undefined`:

```
let limpio = array.filter((el) => el !== undefined);
console.log(limpio); // [1, 3]
```

5. Buenas Prácticas Generales

1. **Usa nombres descriptivos:**
    - En lugar de `arr`, usa `nombres` o `numeros` según el contexto.
2. **Prefiere métodos inmutables:**
    - Si necesitas modificar el array, considera crear una copia antes.
3. **Evita modificar arrays mientras los iteras:**
    - Esto puede generar comportamientos inesperados.

```
let numeros = [1, 2, 3];
numeros.forEach((num) => {
  if (num === 2) {
    numeros.push(4); // Evitar mutaciones dentro de bucles
  }
});
```

4. **Manejo de arrays muy grandes:**
    - Considera usar estructuras de datos como `Map` o `Set` si necesitas optimizar búsquedas o evitar duplicados.