

## Funciones y callback


Además de map, hay varias otras funciones en JavaScript que utilizan callbacks y son comúnmente usadas para manipulación de arrays. Entre las más importantes se encuentran:

- 1) filter
- 2) reduce
- 3) forEach
- 4) some
- 5) every
- 6) find
- 7) findIndex
- 8) sort

## 1. `filter`

La función `filter` crea un nuevo array con todos los elementos que pasan la prueba implementada por la función dada.

js


 Copiar código

```
const numeros = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10];  
const esPar = x => x % 2 === 0;  
const numerosPares = numeros.filter(esPar);  
console.log(numerosPares); // [2, 4, 6, 8, 10]
```

## 2. `reduce`

La función `reduce` aplica una función a un acumulador y a cada valor de la array (de izquierda a derecha) para reducirlo a un solo valor.

js


 Copiar código

```
const sum = (acc, curr) => acc + curr;  
const total = numeros.reduce(sum, 0);  
console.log(total); // 55
```

## 3. `forEach`

La función `forEach` ejecuta la función indicada una vez por cada elemento del array.

js


 Copiar código

```
numeros.forEach(x => console.log(x * 2)); // Imprime 2, 4, 6, ..., 20
```

#### 4. `some`

La función `some` comprueba si al menos un elemento del array cumple con la condición implementada por la función proporcionada.

js

 Copiar código


```
const mayorQueCinco = x => x > 5;  
const existeMayorQueCinco = numeros.some(mayorQueCinco);  
console.log(existeMayorQueCinco); // true
```



#### 5. `every`

La función `every` comprueba si todos los elementos en el array cumplen con la condición implementada por la función proporcionada.

js

 Copiar código


```
const menorQueOnce = x => x < 11;  
const todosMenoresQueOnce = numeros.every(menorQueOnce);  
console.log(todosMenoresQueOnce); // true
```



#### 6. `find`

La función `find` devuelve el primer elemento del array que cumple con la condición implementada por la función proporcionada.

js

 Copiar código


```
const mayorQueSiete = x => x > 7;  
const primerMayorQueSiete = numeros.find(mayorQueSiete);  
console.log(primerMayorQueSiete); // 8
```



## 7. `findIndex`

La función `findIndex` devuelve el índice del primer elemento del array que cumple con la condición implementada por la función proporcionada. Si no hay ningún elemento que cumpla la condición, devuelve `-1`.

js


 Copiar código

```
const indiceMayorQueSiete = numeros.findIndex(mayorQueSiete);  
console.log(indiceMayorQueSiete); // 7
```

## 8. `sort`

La función `sort` ordena los elementos del array y devuelve el array ordenado. La ordenación no es necesariamente estable. El orden predeterminado es según la posición del valor Unicode de los caracteres, pero se puede proporcionar una función de comparación para definir un criterio de ordenación diferente.

js

 Copiar código

```
const numerosDesordenados = [10, 5, 2, 8, 9, 3];  
const ascendente = (a, b) => a - b;  
numerosDesordenados.sort(ascendente);  
console.log(numerosDesordenados); // [2, 3, 5, 8, 9, 10]
```

.L

filter

Filtrar los números pares de un array.

```
const numeros = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10];

const esPar = x => x % 2 === 0;

const numerosPares = numeros.filter(esPar);

console.log(numerosPares); // [2, 4, 6, 8, 10]
```

### **reduce**

Sumar todos los números de un array.

```
const sum = (acc, curr) => acc + curr;

const total = numeros.reduce(sum, 0);

console.log(total); // 55
```

### **forEach**

Multiplicar por 2 cada elemento de un array y mostrarlo.

```
numeros.forEach(x => console.log(x * 2)); // Imprime 2, 4, 6, ..., 20
```

### **some**

Comprobar si hay algún número mayor que 5.

```
const mayorQueCinco = x => x > 5;
```

```
const existeMayorQueCinco = numeros.some(mayorQueCinco);  
  
console.log(existeMayorQueCinco); // true
```

### **every**

Comprobar si todos los números son menores que 11.

```
const menorQueOnce = x => x < 11;  
  
const todosMenoresQueOnce = numeros.every(menorQueOnce);  
  
console.log(todosMenoresQueOnce); // true
```

### **find**

Encontrar el primer número mayor que 7.

```
const mayorQueSiete = x => x > 7;  
  
const primerMayorQueSiete = numeros.find(mayorQueSiete);  
  
console.log(primerMayorQueSiete); // 8
```

### **findIndex**

Encontrar el índice del primer número mayor que 7.

```
const indiceMayorQueSiete = numeros.findIndex(mayorQueSiete);  
  
console.log(indiceMayorQueSiete); // 7
```

## sort

Ordenar un array de números de menor a mayor.

```
const numerosDesordenados = [10, 5, 2, 8, 9, 3];  
  
const ascendente = (a, b) => a - b;  
  
numerosDesordenados.sort(ascendente);  
  
console.log(numerosDesordenados); // [2, 3, 5, 8, 9, 10]
```

## 1. `filter` - Filtrar Objetos Basados en Múltiples Condiciones

### Ejercicio

Filtrar un array de objetos `personas` para obtener solo aquellos que tienen más de 18 años y viven en "New York".

js

Copiar código

```
function filter(array, callback) {  
  
  let resultado = [];
```



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

  if (callback(array[i])) {

    resultado.push(array[i]);

  }

}

return resultado;

}

const personas = [

  { nombre: "Ana", edad: 25, ciudad: "New York" },

  { nombre: "Pedro", edad: 17, ciudad: "Los Angeles" },

  { nombre: "Luis", edad: 19, ciudad: "New York" },

  { nombre: "Laura", edad: 22, ciudad: "Chicago" },

  { nombre: "Carlos", edad: 16, ciudad: "New York" }

];

const mayoresDeEdadEnNY = persona => persona.edad > 18 && persona.ciudad === "New York";

const resultado = filter(personas, mayoresDeEdadEnNY);
```

```
console.log(resultado);

// [{ nombre: "Ana", edad: 25, ciudad: "New York" }, { nombre: "Luis", edad: 19,
ciudad: "New York" }]
```

## 2. reduce - Calcular Estadísticas de un Array de Objetos

### Ejercicio

Calcular el promedio de edad de un array de objetos `personas` y el total de personas que viven en "New York".

js

Copiar código

```
function reduce(array, callback, inicial) {

  let acumulador = inicial;

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

    acumulador = callback(acumulador, array[i]);

  }

  return acumulador;
}

const personas = [

  { nombre: "Ana", edad: 25, ciudad: "New York" },
```

```
{ nombre: "Pedro", edad: 17, ciudad: "Los Angeles" },

{ nombre: "Luis", edad: 19, ciudad: "New York" },

{ nombre: "Laura", edad: 22, ciudad: "Chicago" },

{ nombre: "Carlos", edad: 16, ciudad: "New York" }

];

const estadisticas = (acc, persona) => {

  acc.totalEdad += persona.edad;

  acc.totalPersonas++;

  if (persona.ciudad === "New York") {

    acc.personasEnNY++;

  }

  return acc;

};

const inicial = { totalEdad: 0, totalPersonas: 0, personasEnNY: 0 };

const resultado = reduce(personas, estadisticas, inicial);

resultado.promedioEdad = resultado.totalEdad / resultado.totalPersonas;
```

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

```
// { totalEdad: 99, totalPersonas: 5, personasEnNY: 3, promedioEdad: 19.8 }
```

### 3. map - Transformar un Array de Objetos

#### Ejercicio

Transformar un array de objetos `personas` en un array de strings que contenga solo los nombres de las personas.

js

Copiar código

```
function map(array, callback) {  
  
  let resultado = [];  
  
  for (let i = 0; i < array.length; i++) {  
  
    resultado.push(callback(array[i]));  
  
  }  
  
  return resultado;  
  
}
```

```
const personas = [
```

```
{ nombre: "Ana", edad: 25, ciudad: "New York" },  
  
{ nombre: "Pedro", edad: 17, ciudad: "Los Angeles" },  
  
{ nombre: "Luis", edad: 19, ciudad: "New York" },  
  
{ nombre: "Laura", edad: 22, ciudad: "Chicago" },  
  
{ nombre: "Carlos", edad: 16, ciudad: "New York" }  
  
];
```

```
const obtenerNombre = persona => persona.nombre;
```

```
const nombres = map(personas, obtenerNombre);
```

```
console.log(nombres);
```

```
// ["Ana", "Pedro", "Luis", "Laura", "Carlos"]
```

## 4. `forEach` - Modificar Directamente los Elementos de un Array de Objetos

### Ejercicio

Agregar una nueva propiedad `esMayorDeEdad` a cada objeto en un array de `personas` indicando si la persona es mayor de edad (`edad > 18`).

```
js
```

Copiar código

```
function forEach(array, callback) {

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

    callback(array[i]);

  }

}

const personas = [

  { nombre: "Ana", edad: 25, ciudad: "New York" },

  { nombre: "Pedro", edad: 17, ciudad: "Los Angeles" },

  { nombre: "Luis", edad: 19, ciudad: "New York" },

  { nombre: "Laura", edad: 22, ciudad: "Chicago" },

  { nombre: "Carlos", edad: 16, ciudad: "New York" }

];

const agregarEsMayorDeEdad = persona => {

  persona.esMayorDeEdad = persona.edad > 18;

};
```

```
forEach(personas, agregarEsMayorDeEdad);

console.log(personas);

// [{ nombre: "Ana", edad: 25, ciudad: "New York", esMayorDeEdad: true },
// { nombre: "Pedro", edad: 17, ciudad: "Los Angeles", esMayorDeEdad: false },
// { nombre: "Luis", edad: 19, ciudad: "New York", esMayorDeEdad: true },
// { nombre: "Laura", edad: 22, ciudad: "Chicago", esMayorDeEdad: true },
// { nombre: "Carlos", edad: 16, ciudad: "New York", esMayorDeEdad: false }]
```

## 5. `some` - Verificar Si Algún Elemento Cumple una Condición

### Ejercicio

Comprobar si al menos una persona en el array `personas` vive en "Chicago".

js

Copiar código

```
const personas = [

  { nombre: "Ana", edad: 25, ciudad: "New York" },

  { nombre: "Pedro", edad: 17, ciudad: "Los Angeles" },

  { nombre: "Luis", edad: 19, ciudad: "New York" },

  { nombre: "Laura", edad: 22, ciudad: "Chicago" },

  { nombre: "Carlos", edad: 16, ciudad: "New York" }
```

```
];

const viveEnChicago = persona => persona.ciudad === "Chicago";

const existePersonaEnChicago = personas.some(viveEnChicago);

console.log(existePersonaEnChicago); // true
```

## 6. every - Verificar Si Todos los Elementos Cumplen una Condición

### Ejercicio

Comprobar si todas las personas en el array `personas` son mayores de edad.

js

Copiar código

```
const personas = [

  { nombre: "Ana", edad: 25, ciudad: "New York" },

  { nombre: "Pedro", edad: 17, ciudad: "Los Angeles" },

  { nombre: "Luis", edad: 19, ciudad: "New York" },

  { nombre: "Laura", edad: 22, ciudad: "Chicago" },

  { nombre: "Carlos", edad: 16, ciudad: "New York" }
```



```
];

const esMayorDeEdad = persona => persona.edad > 18;

const todosSonMayoresDeEdad = personas.every(esMayorDeEdad);

console.log(todosSonMayoresDeEdad); // false
```

## 7. find - Encontrar el Primer Elemento que Cumple una Condición

### Ejercicio

Encontrar la primera persona que vive en "New York".

js

Copiar código

```
const personas = [

  { nombre: "Ana", edad: 25, ciudad: "New York" },

  { nombre: "Pedro", edad: 17, ciudad: "Los Angeles" },

  { nombre: "Luis", edad: 19, ciudad: "New York" },

  { nombre: "Laura", edad: 22, ciudad: "Chicago" },

  { nombre: "Carlos", edad: 16, ciudad: "New York" }
```

```
];

const viveEnNewYork = persona => persona.ciudad === "New York";

const primeraPersonaEnNewYork = personas.find(viveEnNewYork);

console.log(primeraPersonaEnNewYork);

// { nombre: "Ana", edad: 25, ciudad: "New York" }
```

## 8. `findIndex` - Encontrar el Índice del Primer Elemento que Cumple una Condición

### Ejercicio

Encontrar el índice de la primera persona que es menor de edad.

js

Copiar código

```
const personas = [

  { nombre: "Ana", edad: 25, ciudad: "New York" },

  { nombre: "Pedro", edad: 17, ciudad: "Los Angeles" },

  { nombre: "Luis", edad: 19, ciudad: "New York" },

  { nombre: "Laura", edad: 22, ciudad: "Chicago" },
```

```
{ nombre: "Carlos", edad: 16, ciudad: "New York" }  
  
];  
  
const esMenorDeEdad = persona => persona.edad < 18;  
  
const indicePrimeraPersonaMenorDeEdad = personas.findIndex(esMenorDeEdad);  
  
console.log(indicePrimeraPersonaMenorDeEdad); // 1
```

## 9. sort - Ordenar un Array de Objetos

### Ejercicio

Ordenar un array de objetos `personas` por edad de menor a mayor.

js

Copiar código

```
const personas = [  
  
  { nombre: "Ana", edad: 25, ciudad: "New York" },  
  
  { nombre: "Pedro", edad: 17, ciudad: "Los Angeles" },  
  
  { nombre: "Luis", edad: 19, ciudad: "New York" },  
  
  { nombre: "Laura", edad: 22, ciudad: "Chicago" },  
  
  { nombre: "Carlos", edad: 16, ciudad: "New York" }
```

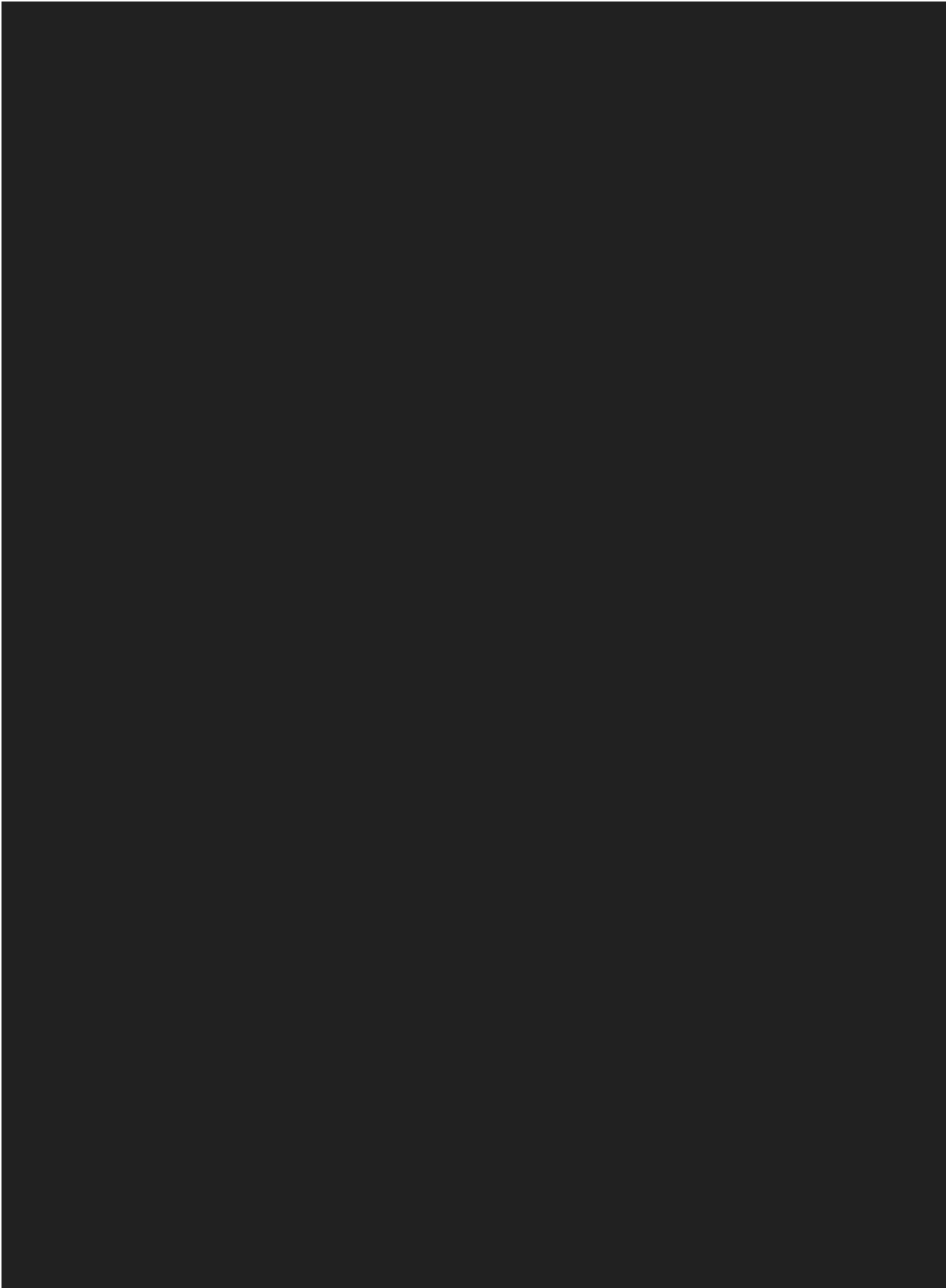
```
];

const ordenarPorEdad = (a, b) => a.edad - b.edad;

const personasOrdenadasPorEdad = personas.sort(ordenarPorEdad);

console.log(personasOrdenadasPorEdad);

// [{ nombre: "Carlos", edad: 16, ciudad: "New York" },
// { nombre: "Pedro", edad: 17, ciudad: "Los Angeles" },
// { nombre: "Luis", edad: 19, ciudad: "New York" },
// { nombre: "Laura", edad: 22, ciudad: "Chicago" },
// { nombre: "Ana", edad: 25, ciudad: "New York" }]
```



## Implementación Mejorada de `every`

En esta versión más compleja de `every`, vamos a permitir que el callback pueda recibir más argumentos, incluyendo el índice del elemento actual y el array completo. También vamos a añadir la posibilidad de usar un objeto `thisArg` para establecer el contexto del callback.

```
function every(array, callback, thisArg) {  
  
  // Crear función every con tres parámetros (array, callback, thisArg)  
  
  // array: el array a evaluar  
  
  // callback: la función que decide si un elemento pasa la condición  
  
  // thisArg: el valor a utilizar como "this" dentro del callback  
  
  
  for (let i = 0; i < array.length; i++) {  
  
    // Recorrer cada elemento en el array de entrada usando un bucle for  
  
    // i es el índice del elemento actual  
  
  
  
    // Verificar si se proporcionó un objeto thisArg y establecer el contexto del callback  
  
    const contexto = thisArg ? thisArg : undefined;  
  
    const resultadoCallback = callback.call(contexto, array[i], i, array);  
  
  }  
}
```

```
// Si el callback devuelve false para el elemento actual

if (!resultadoCallback) {

    return false;

    // Devolver false inmediatamente si no se cumple la condición

}

}

// Si todas las llamadas al callback devolvieron true, devolver true al final

return true;

}

// Ejemplo de uso con objetos complejos

const estudiantes = [

    { nombre: 'Juan', edad: 20, aprobado: true },

    { nombre: 'María', edad: 22, aprobado: false },

    { nombre: 'Pedro', edad: 19, aprobado: true }

];

// Función de callback que verifica si un estudiante está aprobado
```

```
function estaAprobado(estudiante) {  
  
    return estudiante.aprobado;  
  
}  
  
const todosAprobados = every(estudiantes, estaAprobado);  
  
console.log(todosAprobados); // false, porque María no está aprobada  
  
// Ejemplo de uso con thisArg para establecer el contexto del callback  
  
const contextoEjemplo = {  
  
    minEdad: 18,  
  
    maxEdad: 25,  
  
    verificarEdad: function (estudiante) {  
  
        return estudiante.edad >= this.minEdad && estudiante.edad <= this.maxEdad;  
  
    }  
  
};  
  
const todosVerificanEdad = every(estudiantes, contextoEjemplo.verificarEdad,  
contextoEjemplo);  
  
console.log(todosVerificanEdad); // true, todos los estudiantes están dentro del rango de edad
```



## Explicación y Documentación

### 1. Definición de la Función `every`:

- La función `every` toma tres parámetros: `array` (el array a evaluar), `callback` (la función que decide si un elemento pasa la condición) y `thisArg` (el valor a utilizar como "this" dentro del callback).

### 2. Recorrido del Array:

- Se utiliza un bucle `for` para recorrer cada elemento del array de entrada.
- Se verifica si se proporcionó un objeto `thisArg` y se establece el contexto del callback según sea necesario.

### 3. Ejecución del Callback:

- Se llama al callback con tres argumentos: el elemento actual, su índice y el array completo.
- Se almacena el resultado del callback en `resultadoCallback`.

### 4. Verificación de Resultados:

- Si el resultado del callback es `false` para algún elemento, se devuelve `false` inmediatamente.

### 5. Devolver Resultado Final:

- Si todas las llamadas al callback devolvieron `true`, se devuelve `true` al final.

## Ejercicio y Documentación Completa

El ejercicio modificado de `every` permite una mayor flexibilidad al permitir un contexto para el callback y el uso de más argumentos en este. Además, se incluyen ejemplos de uso más complejos con objetos y contextos específicos.

En JavaScript, `call` es un método que se utiliza para llamar a una función especificando un contexto (`this`) y pasando argumentos a esa función. La sintaxis general de `call` es la siguiente:

```
funcion.call(thisArg, arg1, arg2, ...)
```

- `funcion`: La función que se va a llamar.
- `thisArg`: El valor que se utilizará como contexto (`this`) dentro de la función `funcion`.
- `arg1, arg2, ...`: Los argumentos que se pasarán a la función `funcion`.

En el contexto de tu línea de código:

```
const resultadoCallback = callback.call(contexto, array[i], i, array);
```

- `callback`: Es la función que se va a llamar.
- `contexto`: Es el valor que se utilizará como contexto (`this`) dentro de la función `callback`.
- `array[i], i, array`: Son los argumentos que se pasarán a la función `callback`.

Entonces, en esa línea de código, se está llamando a la función `callback` y se especifica `contexto` como el contexto (`this`) dentro de esa llamada, además de pasarle los argumentos `array[i], i` y `array`. Este enfoque es útil cuando se desea ejecutar una función en un contexto específico y pasarle argumentos de manera explícita.

DROP WHILE

## Explicación y Documentación

### 1. Definición de la Función `dropWhile`:

- La función `dropWhile` toma tres parámetros: `array` (el array a evaluar), `callback` (la función que decide si un elemento cumple la condición) y `thisArg` (el valor a utilizar como "this" dentro del callback).

### 2. Inicialización:

- Se inicializa un array vacío `newArray` para almacenar los elementos que no cumplen la condición.
- Se utiliza una variable `dropping` para controlar cuándo dejar de descartar elementos.

### 3. Recorrido del Array:

- Se utiliza un bucle `for` para recorrer cada elemento del array de entrada.
- Se verifica si se proporcionó un objeto `thisArg` y se establece el contexto del callback según sea necesario.

### 4. Ejecución del Callback:

- Se llama al callback con tres argumentos: el elemento actual, su índice y el array completo.
- Se almacena el resultado del callback en `cumpleCondicion`.

### 5. Verificación de Resultados:

- Si el resultado del callback es `false` y se está en modo `dropping`, se cambia `dropping` a `false`.
- Si `dropping` es `false`, se agregan los elementos al nuevo array `newArray`.

### 6. Devolver Resultado Final:

- Se devuelve el nuevo array con los elementos que no cumplieron la condición.

## Ejercicio Complejo

En este ejercicio, la función `dropWhile` permite descartar elementos de un array hasta que se encuentre uno que no cumpla con una condición compleja, con la capacidad de utilizar un contexto (`thisArg`) para el callback. Esto es útil para filtros avanzados en arrays de objetos complejos y cuando las condiciones de filtrado son dinámicas y específicas. Vamos a desglosar por qué se necesitan estas dos variables `newArray` y `dropping` en la función `dropWhile`.

## Variables

### 1. `newArray`:

- Propósito: Almacenar los elementos que no cumplen la condición.
- Función: Se usa para construir y retornar el array final que contiene los elementos a partir del primer elemento que no cumple con la condición impuesta por el callback.
- Explicación: Necesitamos un nuevo array porque `dropWhile` está diseñado para excluir los primeros elementos del array original que cumplen con la condición del callback. Así, `newArray` almacenará únicamente los elementos restantes después de encontrar el primer elemento que no cumple con la condición.

### 2. `dropping`:

- Propósito: Controlar cuándo dejar de descartar elementos.
- Función: Actúa como un interruptor para determinar si seguimos descartando elementos del array original. Se inicializa como `true` para indicar que estamos en modo de "descartar" hasta que encontremos un elemento que no cumpla con la condición.
- Explicación: Necesitamos esta variable porque una vez que encontramos el primer elemento que no cumple la condición, dejamos de descartar elementos y comenzamos a agregarlos a `newArray`. Sin esta variable, no podríamos gestionar correctamente cuándo comenzar a agregar elementos al nuevo array.

## Explicación y Documentación

#### 1. Definición de la Función `takeWhile`:

- La función `takeWhile` toma tres parámetros: `array` (el array a evaluar), `callback` (la función que decide si un elemento cumple la condición) y `thisArg` (el valor a utilizar como "this" dentro del callback).

#### 2. Inicialización:

- Se inicializa un array vacío `newArray` para almacenar los elementos que cumplen la condición del callback.

#### 3. Recorrido del Array:

- Se utiliza un bucle `for` para iterar a través de cada elemento del array de entrada.
- Se verifica si se proporcionó un objeto `thisArg` y se establece el contexto del callback según sea necesario.

#### 4. Ejecución del Callback:

- Se llama al callback con múltiples argumentos: el elemento actual, su índice y el array completo.
- `cumpleCondicion` almacenará el resultado del callback.

#### 5. Verificación de Resultados:

- Si el resultado del callback es `false`, se rompe el bucle y se deja de agregar elementos al nuevo array `newArray`.
- Si el resultado del callback es `true`, el elemento actual se agrega a `newArray`.

#### 6. Devolver el Resultado:

- Finalmente, se retorna `newArray`, que contiene todos los elementos que cumplen la condición hasta el primer elemento que no la cumple.

## Explicación del Flujo

#### 1. Inicialización:

- `newArray` se crea como un array vacío que almacenará los elementos que cumplen la condición del callback.

## 2. Recorrido del Array:

- Usamos un bucle `for` para iterar a través de cada elemento del array.

## 3. Ejecución del Callback:

- Se llama al callback con el contexto apropiado y se pasan el elemento actual, su índice y el array completo.
- `cumpleCondicion` almacenará el resultado del callback.

## 4. Condiciones de Agregación:

- Si el resultado del callback es `false`, se rompe el bucle.
- Si el resultado del callback es `true`, el elemento actual se agrega a `newArray`.

## 5. Devolver el Resultado:

- Finalmente, se retorna `newArray`, que contiene todos los elementos que cumplen la condición hasta el primer elemento que no la cumple.

Este código es un ejemplo de cómo usar JavaScript para realizar solicitudes a una API y procesar los datos obtenidos. Vamos a analizarlo paso a paso:

1. `filtrar`: Esta es una función de flecha que toma un parámetro `x` y verifica si el atributo `name` de ese objeto es igual a "Evaluacion". Retorna `true` si la condición se cumple y `false` en caso contrario.
2. `(async () => { ... })()`: Esta es una función asíncrona autoejecutable, lo que significa que se ejecuta inmediatamente después de ser definida. El uso de `async` permite el uso de `await` dentro de la función para esperar a que las promesas se resuelvan de forma asíncrona.
3. `await fetch("user.json")`: Realiza una solicitud para obtener el contenido del archivo "user.json" de forma asíncrona. La palabra clave `await` pausa la ejecución de la función hasta que la promesa devuelta por `fetch` se resuelva.
4. `await response.json()`: Convierte la respuesta obtenida en el paso anterior a formato JSON para poder trabajar con ella como un objeto JavaScript.
5. `await fetch(https://api.github.com/users/\${user.name}/repos`)`: Realiza una solicitud a la API de GitHub para obtener los repositorios del usuario cuyo nombre se encuentra en el archivo "user.json". Aquí se utiliza la interpolación de cadenas (`${...}`) para incluir dinámicamente el nombre de usuario en la URL de la solicitud.
6. `await respuestaGithub.json()`: Convierte la respuesta de la API de GitHub a formato JSON para poder trabajar con ella como un objeto JavaScript.
7. `usuariogithub.forEach(element => { ... })`: Itera sobre el array de repositorios obtenidos y realiza una acción para cada elemento del array. En este caso, verifica si el nombre del repositorio es "Evaluacion" y lo imprime en la consola si es así.




8. Opcionalmente, se muestra cómo se podría usar la función `filtrar` para obtener el mismo resultado que el bucle `forEach`. Esta es una forma más declarativa de filtrar elementos en un array.
9. `console.log(...)`: Se utiliza para imprimir información en la consola del navegador. En este caso, se imprimen los repositorios que cumplen con la condición especificada.

En resumen, este código carga información de un archivo JSON local y luego utiliza esa información para realizar una solicitud a la API de GitHub, filtrar los repositorios del usuario y mostrar los que tienen el nombre "Evaluacion".

Además de `fetch`, hay varias funciones y conceptos que se utilizan comúnmente junto con `then`, `catch` o `async/await` para trabajar con promesas en JavaScript:

1. `then()`: Se usa para manejar el resultado exitoso de una promesa. La función que se pasa a `then` se ejecutará cuando la promesa se resuelva exitosamente, y recibirá el valor resuelto como argumento.


javascript

 Copiar código

```
fetch("https://api.example.com/data")
  .then(response => response.json())
  .then(data => console.log(data))
  .catch(error => console.error("Error:", error));
```

2. `catch()`: Se utiliza para manejar errores que puedan ocurrir durante la ejecución de una promesa. La función pasada a `catch` se ejecutará si la promesa es rechazada.


javascript

 Copiar código

```
fetch("https://api.example.com/data")
  .then(response => response.json())
  .then(data => console.log(data))
  .catch(error => console.error("Error:", error));
```

3. `async` y `await`: Estas son palabras clave que se utilizan para trabajar de manera más sincrónica con promesas. `async` se coloca antes de una función para indicar que la función devuelve una promesa. `await` se usa dentro de una función `async` para esperar que una promesa se resuelva antes de continuar con la ejecución.

javascript

 Copiar código


```
async function fetchData() {
  try {
    let response = await fetch("https://api.example.com/data");
    let data = await response.json();
    console.log(data);
  } catch (error) {
    console.error("Error:", error);
  }
}

fetchData();
```



4. `Promise.resolve()` y `Promise.reject()`: Estas funciones estáticas de la clase `Promise` se utilizan para crear promesas resueltas y rechazadas respectivamente.

javascript


 Copiar código

```
let resolvedPromise = Promise.resolve("Valor resuelto");
let rejectedPromise = Promise.reject("Error: algo salió mal");

resolvedPromise.then(value => console.log(value)); // "Valor resuelto"
rejectedPromise.catch(error => console.error(error)); // "Error: algo salió mal"
```

5. `Promise.all()` y `Promise.race()`: Estas son funciones estáticas de la clase `Promise` que se utilizan para trabajar con múltiples promesas. `Promise.all()` espera a que todas las promesas dadas se resuelvan o alguna de ellas se rechace, mientras que `Promise.race()` resuelve o rechaza la primera promesa que se resuelva o rechace, ignorando las demás.

javascript

 Copiar código

```
let promise1 = fetch("https://api.example.com/data1");
let promise2 = fetch("https://api.example.com/data2");

Promise.all([promise1, promise2])
  .then(responses => Promise.all(responses.map(response => response.json())))
  .then(data => console.log(data))
  .catch(error => console.error("Error:", error));

Promise.race([promise1, promise2])
  .then(response => response.json())
  .then(data => console.log(data))
  .catch(error => console.error("Error:", error));
```

## Fetch y .json()

Cuando usamos `fetch` para obtener un archivo JSON del servidor, lo que obtenemos es una "respuesta" del servidor. Esta respuesta contiene información sobre el archivo JSON que hemos solicitado, como su contenido y algunos detalles adicionales.

La variable que usamos para almacenar esta respuesta se llama `response`. Pero la variable `response` en sí misma no es el archivo JSON que queremos obtener. En cambio, es como un "contenedor" que contiene ese archivo JSON.

Para acceder realmente al contenido del archivo JSON (es decir, a los datos que contiene el archivo JSON), utilizamos el método `.json()` que está disponible en el objeto `response`. Al hacer `response.json()`, le estamos diciendo al navegador: "Quiero tomar el contenido del archivo JSON que está dentro de esta respuesta y convertirlo en un objeto JavaScript que pueda usar en mi código".

Entonces, `response.json()` no está devolviendo la variable `response` como un archivo `.json`. Más bien, está tomando el contenido del archivo JSON que está dentro de `response` y lo está convirtiendo en un objeto JavaScript que podemos usar para trabajar con los datos del archivo JSON en nuestro código.

- Cuando usamos `array.length`, estamos utilizando una propiedad del objeto `array` para obtener información sobre el número de elementos que tiene el array. Esta propiedad nos da acceso a los datos que están dentro del array, en este caso, nos da la cantidad de elementos que contiene.
- Del mismo modo, cuando usamos `response.json()` en una solicitud Fetch para obtener un archivo JSON del servidor, estamos utilizando un método del objeto

`response` para obtener el contenido del archivo JSON y convertirlo en un objeto JavaScript que podamos utilizar en nuestro código. Este método nos da acceso a los datos que están dentro del archivo JSON que hemos recibido del servidor.

Así que sí, puedes equiparar `response.json()` con `array.length` en términos de acceso a los datos que están dentro de un objeto (ya sea un array o un archivo JSON).

Correcto, la forma en que JavaScript maneja la asincronía y el uso de Promesas está relacionada con el hecho de que JavaScript es de un solo hilo (o más precisamente, tiene un modelo de concurrencia basado en eventos y bucle de eventos).

En JavaScript, cuando realizamos operaciones asincrónicas como solicitudes a servidores (como `fetch`), lecturas de archivos, temporizadores, etc., estas operaciones no bloquean la ejecución del código. En lugar de eso, JavaScript continúa ejecutando otras tareas mientras espera que estas operaciones asincrónicas se completen.

Las Promesas son una forma de manejar estas operaciones asincrónicas de manera más organizada y fácil de entender. Cuando usamos `.then()`, estamos diciendo a JavaScript que ejecute cierto código después de que una operación asincrónica se haya completado y la Promesa se haya resuelto.

En cuanto a la especificación de `data` como representante del objeto que contiene los datos del archivo JSON en el código, esto se realiza de forma implícita mediante el uso de la sintaxis de Promesas y el encadenamiento de métodos `.then()`. Al encadenar `.then(data => { ... })` después de una operación que devuelve una Promesa (como `response.json()`), estamos estableciendo que `data` será el resultado de esa Promesa una vez que se resuelva.

Por lo tanto, no es necesario especificar explícitamente que `data` representará el objeto de datos del archivo JSON en el código. Esto se determina automáticamente en función de cómo se encadenan las Promesas y cómo se manejan los resultados en las funciones de los `.then()`.