

Unidad 6: Funciones del Orden Superior

# **¿Qué son las funciones del orden superior?**

En JavaScript, las funciones del orden superior son funciones que pueden recibir otras funciones como argumentos, retornar funciones como resultado, o ambas cosas. Este concepto es fundamental en la programación funcional y es una característica poderosa de JavaScript, que permite crear código más modular, reutilizable y flexible.

### **Recibir funciones como parámetros**

Una función de orden superior puede aceptar una o más funciones como parámetros. Esto permite que la función de orden superior pueda ejecutar la función pasada como argumento dentro de su propia lógica, permitiendo personalizar el comportamiento de esa función de acuerdo a las necesidades.

**Ejemplo:**

| const numeros = [1, 2, 3, 4];  function porCadaUno(array, fn) {  for (let i = 0; i < array.length; i++) {  fn(array[i]);  }  }  porCadaUno(numeros, console.log);  // 1  // 2  // 3  // 4 |
| --- |

En este ejemplo, la función porCadaUno recibe un array y una función (fn) como parámetros. Luego, itera sobre cada elemento del array y ejecuta la función pasada (console.log en este caso) sobre cada elemento.

### **Retornar funciones**

Además de recibir funciones como parámetros, una función de orden superior puede devolver una nueva función. Esta técnica permite crear funciones más especializadas y configurables.

**Ejemplo:**

| function mayorQue(n) {  return function(m) {  return m > n;  }  }  let mayorQueDiez = mayorQue(10);  console.log(mayorQueDiez(12)); // true  console.log(mayorQueDiez(8)); // false |
| --- |

En este caso, la función mayorQue retorna una nueva función que compara su argumento m con el valor n original. La función mayorQueDiez es un ejemplo de cómo crear funciones especializadas a partir de funciones generales.

### **Importancia de las funciones del orden superior**

Las funciones del orden superior son cruciales en JavaScript por varias razones:

1. **Modularidad**: Permiten dividir el código en componentes más pequeños y manejables.
2. **Reutilización de código**: Facilitan la creación de funciones genéricas que pueden ser reutilizadas en diferentes contextos.
3. **Flexibilidad**: Al permitir la personalización del comportamiento de una función mediante la inyección de funciones como parámetros, se puede crear código más flexible y adaptable.
4. **Programación funcional**: Son la base para muchas técnicas de programación funcional, como mapear, filtrar o reducir colecciones de datos.

### **Métodos comunes que usan funciones del orden superior**

JavaScript proporciona varios métodos que utilizan funciones del orden superior para manipular arrays:

* **forEach**: Itera sobre un array y ejecuta una función por cada elemento.
* **map**: Crea un nuevo array con los resultados de la función aplicada a cada elemento del array original.
* **filter**: Crea un nuevo array con todos los elementos que pasan una prueba definida por la función.
* **reduce**: Ejecuta una función reductora sobre cada elemento del array, resultando en un único valor de retorno.
* **find**: Retorna el primer elemento que cumple con una condición definida en la función.
* **some**: Verifica si al menos un elemento en el array cumple con la condición implementada por la función.

**Ejemplo con map:**

| const numeros = [1, 2, 3, 4];  const duplicados = numeros.map(function(num) {  return num \* 2;  });  console.log(duplicados); // [2, 4, 6, 8] |
| --- |

En este ejemplo, el método map aplica la función que duplica los valores a cada elemento del array original, retornando un nuevo array con los resultados.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

# **Métodos de Búsqueda**

## **forEach: Iterando Arrays**

### **¿Qué es el método forEach?**

El método forEach es un método incorporado en los arrays de JavaScript que permite iterar sobre cada uno de los elementos de un array, ejecutando una función proporcionada en cada elemento. Este método es especialmente útil cuando se desea realizar una operación en cada elemento de un array, como imprimir valores, modificar elementos o realizar cálculos, sin la necesidad de escribir un bucle for tradicional.

**¿Cómo se utiliza el método forEach?**

El forEach recibe una función como argumento. Esta función se ejecutará una vez por cada elemento del array, y recibe como argumento el valor del elemento actual en la iteración. Opcionalmente, esta función también puede recibir dos argumentos adicionales: el índice del elemento actual y el array completo.

**Sintaxis básica:**

| array.forEach(function(elemento, indice, array) {  // Código a ejecutar por cada elemento  }); |
| --- |

* **elemento**: Es el valor del elemento actual en el array.
* **indice** (opcional): Es el índice del elemento actual en el array.
* **array** (opcional): Es el array completo que está siendo iterado.

### **Ejemplos básicos**

1. **Iteración simple sobre un array:**

| const numeros = [1, 2, 3, 4, 5];  numeros.forEach(function(numero) {  console.log(numero);  }); |
| --- |

**Salida:**

| 1  2  3  4  5 |
| --- |

En este ejemplo, el método forEach recorre el array numeros e imprime cada uno de los elementos en la consola.

1. **Uso del índice en la iteración:**

| const frutas = ['Manzana', 'Banana', 'Cereza'];  frutas.forEach(function(fruta, indice) {  console.log(`Índice ${indice}: ${fruta}`);  }); |
| --- |

**Salida:**

| Índice 0: Manzana  Índice 1: Banana  Índice 2: Cereza |
| --- |

Aquí, además del valor del elemento, también se utiliza el índice para mostrar la posición de cada fruta en el array.

1. **Modificación de elementos en un array (aunque no es el uso principal de forEach):**

| const numeros = [1, 2, 3, 4];  const duplicados = [];  numeros.forEach(function(numero) {  duplicados.push(numero \* 2);  });  console.log(duplicados); // [2, 4, 6, 8] |
| --- |

En este ejemplo, se utiliza forEach para crear un nuevo array duplicados, donde cada número del array original numeros se multiplica por 2.

### **Consideraciones importantes**

* **No modifica el array original:** A diferencia de otros métodos de array como map, forEach no retorna un nuevo array ni modifica el array original (aunque sí puede afectar otros arrays o variables externas si se utilizan dentro de la función).
* **No retorna un valor:** El método forEach siempre retorna undefined. Si se necesita crear un nuevo array a partir de la transformación de los elementos, métodos como map son más apropiados.
* **No se puede interrumpir:** A diferencia de un bucle for, no se puede salir o romper un forEach antes de que haya terminado de iterar sobre todos los elementos. Si se necesita esa funcionalidad, es mejor utilizar un bucle for o for...of.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

# 

# **Métodos de Transformación**

## **filter: Filtrando arrays**

### **¿Qué es el método filter?**

El método filter es una función de orden superior en JavaScript que permite crear un nuevo array a partir de un array existente, incluyendo solamente aquellos elementos que cumplan con una condición específica. Es una herramienta muy poderosa cuando se necesita filtrar datos de una colección, ya sea para reducir la cantidad de datos o para enfocarse en un subconjunto específico de información.

**¿Cómo funciona el método filter?**

El método filter recibe una función como argumento. Esta función se ejecuta en cada uno de los elementos del array y debe devolver un valor booleano (true o false). Si la función devuelve true, el elemento actual se incluye en el nuevo array; si devuelve false, el elemento se excluye.

**Sintaxis básica:**

| const nuevoArray = array.filter(function(elemento, indice, array) {  // Condición que debe cumplirse  }); |
| --- |

* **elemento**: El elemento actual que se está procesando en el array.
* **indice** (opcional): El índice del elemento actual en el array.
* **array** (opcional): El array sobre el cual se está aplicando el método filter.

### **Ejemplos básicos**

1. **Filtrar números pares de un array:**

| const numeros = [1, 2, 3, 4, 5, 6];  const numerosPares = numeros.filter(function(numero) {  return numero % 2 === 0;  });  console.log(numerosPares); // [2, 4, 6] |
| --- |

En este ejemplo, el método filter crea un nuevo array numerosPares, que contiene solo los números que son divisibles por 2, es decir, los números pares.

1. **Filtrar objetos en un array:**

| const cursos = [  { nombre: 'JavaScript', duracion: 8 },  { nombre: 'React', duracion: 6 },  { nombre: 'Angular', duracion: 10 }  ];  const cursosLargos = cursos.filter(function(curso) {  return curso.duracion > 7;  });  console.log(cursosLargos);  // [{ nombre: 'JavaScript', duracion: 8 }, { nombre: 'Angular', duracion: 10 }] |
| --- |

En este caso, el método filter se usa para crear un nuevo array cursosLargos que solo incluye los cursos cuya duración es mayor que 7 semanas.

1. **Filtrar elementos que contienen una subcadena:**

| const palabras = ['manzana', 'banana', 'cereza', 'damasco'];  const palabrasConA = palabras.filter(function(palabra) {  return palabra.includes('a');  });  console.log(palabrasConA); // ['manzana', 'banana', 'cereza', 'damasco'] |
| --- |

Este ejemplo muestra cómo filter puede ser utilizado para crear un nuevo array palabrasConA que contiene solo las palabras que incluyen la letra "a".

**Aplicaciones prácticas del método filter**

El método filter es extremadamente útil en muchos contextos diferentes:

* **Filtrar datos de una API**: Cuando se obtienen datos de una API, como una lista de productos, se puede usar filter para mostrar solo los productos que están en stock, que están en oferta, o que cumplen con alguna otra condición.
* **Filtrar resultados de búsqueda**: En aplicaciones que permiten a los usuarios buscar entre múltiples ítems, filter se puede utilizar para mostrar solo los resultados que coinciden con los criterios de búsqueda.
* **Filtrar elementos en formularios**: Si un formulario tiene múltiples entradas o respuestas, filter puede ayudar a identificar y trabajar solo con aquellas que son válidas o relevantes.

**Consideraciones importantes**

* **No modifica el array original**: El método filter no altera el array original. En su lugar, retorna un nuevo array que contiene los elementos que cumplen con la condición especificada.
* **Retorna un array**: Incluso si ningún elemento cumple la condición, filter siempre devolverá un array (posiblemente vacío).
* **Encadenamiento**: filter se puede encadenar con otros métodos como map o reduce para realizar operaciones más complejas en los datos.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

# **Ejemplos Avanzados**

## **reduce: Acumulando valores**

### **¿Qué es el método reduce?**

El método reduce es una poderosa función de orden superior en JavaScript que se utiliza para reducir un array a un único valor. Este método ejecuta una función reductora en cada elemento del array (de izquierda a derecha), acumulando el resultado en un valor único. reduce es especialmente útil cuando se necesita realizar operaciones que involucran la acumulación de datos, como la suma de todos los elementos de un array o la combinación de múltiples valores en un solo resultado.

**¿Cómo se utiliza el método reduce?**

El método reduce recibe dos parámetros principales:

1. **Función reductora**: Una función que se ejecuta en cada elemento del array. Esta función recibe cuatro argumentos:
   * **acumulador**: El valor acumulado hasta el momento.
   * **valorActual**: El elemento actual que se está procesando en el array.
   * **indiceActual** (opcional): El índice del elemento actual.
   * **array** (opcional): El array original sobre el cual se está aplicando el método reduce.
2. **Valor inicial** (opcional): Un valor inicial para el acumulador. Si no se proporciona, reduce usará el primer elemento del array como el valor inicial y comenzará a partir del segundo elemento.

**Sintaxis básica:**

| array.reduce(function(acumulador, valorActual, indiceActual, array) {  // Código para combinar acumulador y valorActual  }, valorInicial); |
| --- |

### **Ejemplos de uso común del método reduce**

1. **Sumar todos los elementos de un array:**

| const numeros = [1, 2, 3, 4, 5];  const suma = numeros.reduce(function(acumulador, valorActual) {  return acumulador + valorActual;  }, 0);  console.log(suma); // 15 |
| --- |

En este ejemplo, reduce suma todos los elementos del array numeros. El acumulador comienza en 0 (el valor inicial proporcionado) y se le suma cada elemento del array, resultando en 15.

1. **Encontrar el valor máximo en un array:**

| const numeros = [10, 5, 20, 15];  const maximo = numeros.reduce(function(acumulador, valorActual) {  return Math.max(acumulador, valorActual);  });  console.log(maximo); // 20 |
| --- |

Aquí, reduce se utiliza para encontrar el valor máximo en el array. Compara el acumulador con el valor actual y mantiene el mayor de los dos en cada iteración.

1. **Contar la cantidad de ocurrencias de elementos en un array:**

| const frutas = ['manzana', 'banana', 'naranja', 'manzana', 'naranja', 'banana', 'banana'];  const conteoFrutas = frutas.reduce(function(conteo, fruta) {  conteo[fruta] = (conteo[fruta] || 0) + 1;  return conteo;  }, {});  console.log(conteoFrutas);  // { manzana: 2, banana: 3, naranja: 2 } |
| --- |

En este ejemplo, reduce se utiliza para contar la cantidad de veces que aparece cada fruta en el array. El acumulador es un objeto donde se almacenan los conteos de cada fruta.

### **Aplicaciones avanzadas del método reduce**

* **Transformación de estructuras de datos**: reduce es muy útil para transformar un array en otro tipo de estructura de datos, como un objeto o incluso otro array más complejo.
* **Agrupación de datos**: Puedes usar reduce para agrupar elementos de un array en categorías, generando un objeto donde las claves son los nombres de las categorías y los valores son arrays de elementos que pertenecen a esas categorías.
* **Encadenamiento con otros métodos**: reduce puede ser utilizado junto con otros métodos de arrays, como map o filter, para realizar operaciones complejas en un solo paso.

### **Consideraciones importantes**

* **El valor inicial es importante**: Siempre considera el valor inicial del acumulador, ya que afecta cómo se procesa el array. Si no se proporciona, el primer elemento del array se usará como valor inicial, lo que puede llevar a errores si el array está vacío.
* **Complexidad y legibilidad**: Aunque reduce es muy poderoso, su uso puede hacer que el código sea más difícil de entender. Es importante asegurarse de que el uso de reduce esté justificado y que el código siga siendo legible para otros desarrolladores.